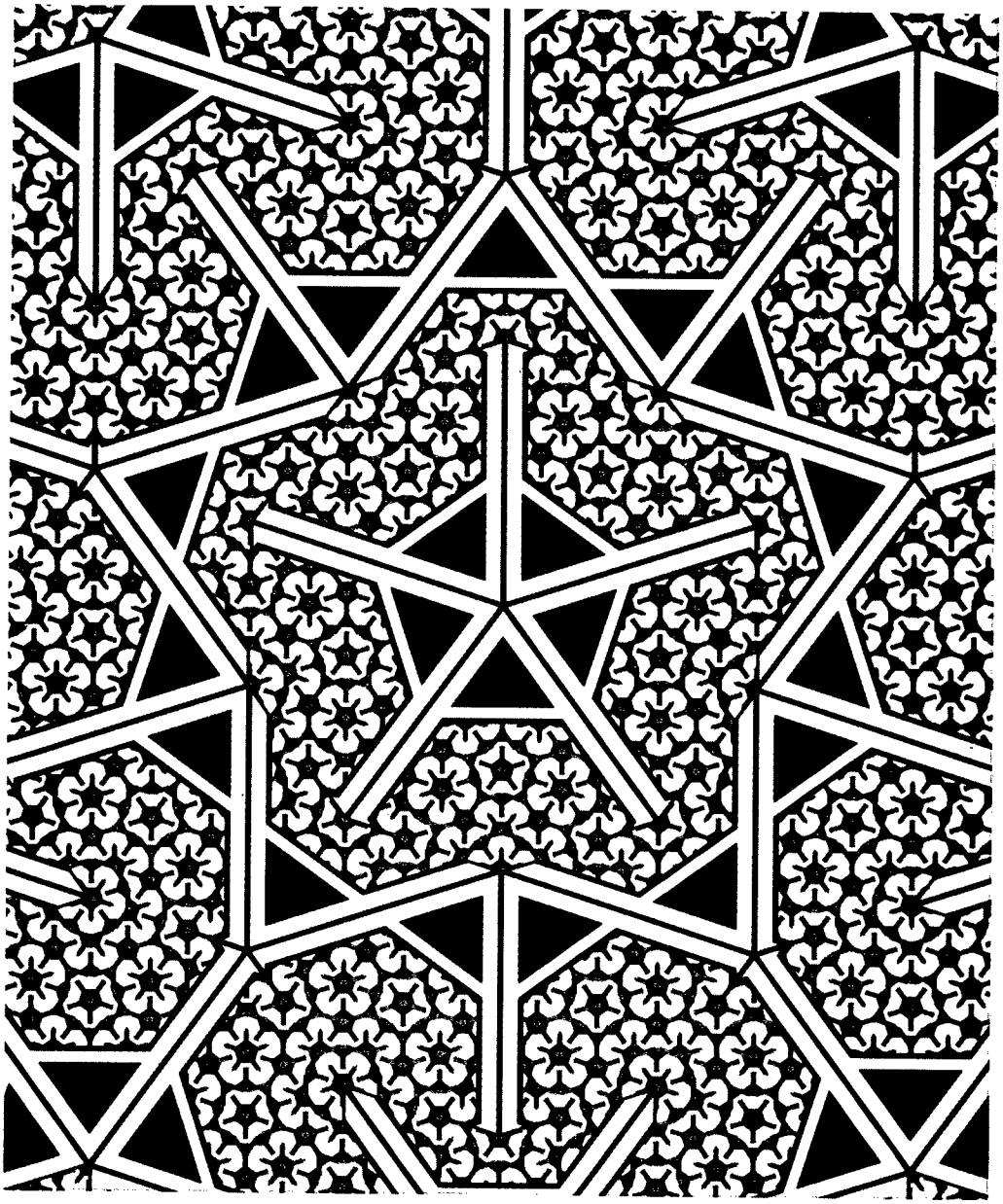
PROCEEDINGS

SIXTEENTH ANNUAL IEEE SYMPOSIUM ON Logic in Computer Science

16-19 JUNE 2001

BOSTON, MASSACHUSETTS



Sponsored by IEEE Computer Society Technical Committee on Mathematical Foundations of Computing

IEEE COMPUTER SOCIETY

20020408 022

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

Public Reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimates or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188,) Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 31, 2001	Final

4. TITLE AND SUBTITLE
2001 IEEE Conference on Logic and Computer Science (LICS 2001)

5. FUNDING NUMBERS
N00014-01-1-0568

6. AUTHOR(S)
Joseph Halpern (Editor)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IEEE LICS

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Ralph Wachter
ONR
6 Ballston Tower One
800 North Quincy Street
Arlington, VA 22217

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The LICS Symposium is an annual international forum on theoretical and practical topics in computer science that relate to logic in a broad sense. Topics of interest include: automata theory, category theory, concurrency, constraint programming, database theory, domain theory, finite model theory, formal methods, hybrid systems, language calculi, linear logic, complexity, artificial intelligence, logic programming, modal and temporal logics, model checking, semantics, security, rewriting, specifications, type theory, and verification.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Logic, Computer Science,automata,language calculi, concurrency, formal methods, model checking, security, specifications, verification	441
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION ON THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used for announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

Block 1. Agency Use Only (Leave blank)

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, and volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s) project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	**PR** - Project
G - Grant	**TA** - Task
PE - Program Element	**WU** - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es) Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(if known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as; prepared in cooperation with....; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NORFORN, REL, ITAR).

- **DOD** - See DoDD 4230.25, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave Blank
- **DOE** - Enter DOE distribution categories from the Standard Distribution for unclassified Scientific and Technical Reports
- **NASA** - Leave Blank.
- **NTIS** - Leave Blank.

Block 13. Abstract. Include a brief (*Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subject in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS *only*).

Block 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (Unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Proceedings

16th Annual IEEE Symposium on
Logic in Computer Science

Proceedings

16th Annual IEEE Symposium on

Logic in Computer Science

16–19 June 2001 • Boston, Massachusetts

Sponsored by

IEEE Computer Society Technical Committee on

Mathematical Foundations of Computing



Los Alamitos, California

Washington • Brussels • Tokyo

Copyright © 2001 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331.

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society, or the Institute of Electrical and Electronics Engineers, Inc.

IEEE Computer Society Order Number PR01281
ISBN 0-7695-1281-X
ISSN: 1043-6871

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1 714 821 8380
Fax: + 1 714 821 4641
http://computer.org/
csbooks@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1 732 981 0060
Fax: + 1 732 981 9667
http://shop.ieee.org/store/
customer-service@ieee.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-ku, Tokyo 107-0062
JAPAN
Tel: + 81 3 3408 3118
Fax: + 81 3 3408 3553
tokyo.ofc@computer.org

Editorial production by A. Denise Williams

Cover graphic design by Alvy Ray Smith

Cover art production by Joseph Daigle/Studio Productions

Printed in the United States of America by The Printing House, Inc.

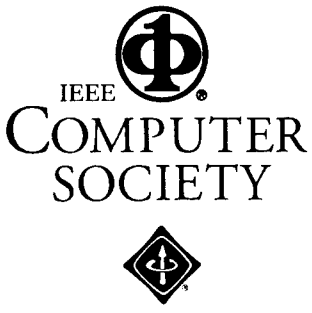


Table of Contents

16th Annual IEEE Symposium on Logic in Computer Science

Foreword x

Conference Organization xi

Additional Reviewers xii

Invited Talk

Chair: Joseph Y. Halpern

Probabilistic Polynomial-Time Precess Calculus and Security Protocol Analysis 3
J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague

Session 1

Chair: Jean-Pierre Jouannaud

Definitions by Rewriting in the Calculus of Constructions 9
F. Blanqui

Deconstructing Shostak 19
H. Rueß and N. Shankar

A Decision Procedure for an Extensional Theory of Arrays 29
A. Stump, C. Barrett, D. Dill, and J. Levitt

On Ordering Constraints for Deduction with Built-In Abelian Semigroups, Monoids and Groups 38
G. Godoy and R. Nieuwenhuis

Invited Talk

Chair: Jean-Pierre Jouannaud

Successive Approximation of Abstract Transition Relations 51
S. Das and D. Dill

Session 2

Chair: Pawel Urzyczyn

A Bound on Attacks on Payment Protocols 61
S. Stoller

A Dichotomy in the Complexity of Propositional Circumscription 71
L. Kirousis and P. Kolaitis

Relating Semantic and Proof-Theoretic Concepts for Polynomial Time Decidability of Uniform Word Problems 81
H. Ganzinger

Session 3

Chair: Radha Jaghadeesan

Semantics of Name and Value Passing 93
M. Fiore and D. Turi

A Fully Abstract Game Semantics of Local Exceptions 105
J. Laird

A Universal Characterization of the Closed Euclidean Interval 115
M. Escardó and A. Simpson

Invited Talk

Chair: Gordon Plotkin

Logician in the Land of OS: Abstract State Machines in Microsoft 129
Y. Gurevich

Session 4

Chair: Michel de Rougemont

Eliminating Definitions and Skolem Functions in First-Order Logic 139
J. Avigad

On the Decision Problem for the Guarded Fragment with Transitivity 147
W. Szwast and L. Tendera

The Hierarchy inside Closed Monadic Σ_1 Collapses on the Infinite Binary Tree 157
A. Arnold, G. Lenzi, and J. Marcinkowski

On Definability of Order in Logic with Choice 167
T. Huuskonen and T. Hyttinen

Invited Talk

Chair: Erich Graedel

The Engineering Challenge for Logic
Wolfgang Thomas

Session 5

Chair: Erich Graedel

A Second-Order System for Polytime Reasoning Using Graedel's Theorem 177
S. Cook and A. Kolokolova

The Crane Beach Conjecture 187
D. Barrington, N. Immerman, C. Lautemann, N. Schweikardt, and D. Thérien

An *n!* Lower Bound on Formula Size 197
M. Adler and N. Immerman

Session 6

Chair: Nevin Heintze

Light Affine Lambda Calculus and Polytime Strong Normalization 209
K. Terui

Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory 221
F. Pfenning

Dependent Types for Program Termination Verification 231
H. Xi

Short Paper Session

Chair: Joseph Y. Halpern

The Dolev-Yao Intruder is the Most Powerful Attacker
I. Cervesato

Semantics of Machine Instructions at Multiple Levels of Abstraction
G. Tan and A. Appel

A Proof-Carrying Authorization System
L. Bauer, M. Schneider, and E. Felten

Recursive Programming Languages for Complexity Classes
E. Covino and G. Pani

Interior-Point Approach to Parity Games
V. Petersson and S. Vorobyov

Recent Progress in Proof Mining
U. Kohlenbach

On the Complexity of Confluence for Ground Rewrite Systems
A. Hayrapetyan and R. Verma

Computing the Density of Regular Languages
M. Bodirsky, M. Gaertner, T. von Oertzen, and J. Schwinghammer

Integrating Simplification Techniques in SAT Algorithms
I. Lynce and J. Marques-Silva

Basic Completion Modulo with Simplification
C. Lynch and C. Scharff

Finite Visit Sequential Deterministic Tree Automata
S. Lindell

Invited Talk

Chair: Ron van der Meyden

Foundational Proof-Carrying Code 247
A. Appel

Session 7

Chair: Parosh Abdulla

Intuitionistic Linear Logic and Partial Correctness 259
D. Kozen and J. Tiuryn

Perturbed Turing Machines and Hybrid Systems 269
E. Asarin and A. Bouajjani

From Verification to Control: Dynamic Programs for Omega-Regular Objectives 279
L. de Alfaro, T. Henzinger, and R. Majumdar

Deterministic Generators and Games for LTL Fragments 291
R. Alur and S. La Torre

Session 8

Chair: Adolfo Piperno

Normalization by Evaluation for Typed Lambda Calculus with Coproducts 303
T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott

Strong Normalisation in the π-Calculus 311
N. Yoshida, M. Berger, and K. Honda

A Symbolic Labelled Transition System for Coinductive Subtyping of $F\mu\leq$ Types 323
A. Jeffrey

A Continuum of Theories of Lambda Calculus without Semantics 334
A. Salibra

Session 9

Chair: Hubert Comon

Relating Levels of the Mu-Calculus Hierarchy and Levels of the Monadic Hierarchy 347
D. Janin and G. Lenzi

Focus Games for Satisfiability and Completeness of Temporal Logic 357
M. Lange and C. Stirling

Safety and Liveness in Branching Time 366
P. Manolios and R. Trefler

Short Papers

Self-Verifying Systems, the Incompleteness Theorem and the Tangibility Reflection Principle
D. Willard

Repairing the Interpolation Theorem in First-Order Modal Logic
C. Areces, P. Blackburn, and M. Marx

A Game involving Epistemic Logic and Probability
A. Pogel, G. Voutsadakis, and M. Gehrke

A Theory of Advanced Transactions in the Situation Calculus
I. Kiringa

Invited Talk

Chair: Michel de Rougemont

Semistructured Data: From Practice to Theory 379
S. Abiteboul

Session 10

Chair: Rance Cleaveland

Synthesizing Distributed Systems 389
O. Kupferman and M. Vardi

Permutation Rewriting and Algorithmic Verification 399
A. Bouajjani, A. Muscholl, and T. Touili

Temporal Logic Query Checking 409
G. Bruns and P. Godefroid

Session 11

Chair: Ron van der Meyden

Typechecking XML Views of Relational Databases 421
N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu

A Model-Theoretic Approach to Regular String Relations 431
M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin

Author Index 441

Foreword

It's hard to believe that this is already the 16th LICS. It doesn't seem all that long (at least to me!) since the conference started. The program chair of the first LICS was Albert Meyer. This year, one of the workshops associated with LICS is the Symposium on Complexity, Logic, and Computation, in honor of Albert.

From the 104 submissions received, the Program Committee selected thirty-six papers. Many worthy abstracts had to be rejected due to the time constraints of the conference. These papers are preliminary reports of ongoing research. Most will appear in more polished and complete form in scientific journals. There are also six invited talks that are represented in the proceedings. Finally, the titles of fifteen short talks are listed. These are mainly announcements: in some cases, full papers are available from the authors; in other cases, the research is so preliminary that there is no paper yet.

Many people put in a great deal of time and effort into selecting the program. First and foremost, there was the Program Committee. These days, program committee meetings are virtual; they are conducted asynchronously by email. That means that "meetings" take place over a 10-day period. Program committee members had to read email at all times of the day just to keep up. Fortunately for me, this was a very active program committee, and they seemed to be willing to do that. Even better, we were able to converge to a program that we were all comfortable with in a remarkably smooth manner. This year we put a special emphasis on having papers where the relevance to computer science was clear and which would be accessible to nonexperts. These proceedings should attest to how well we succeeded.

Another one of our tasks was to choose the best student paper(s) for the Kleene award. This year there are two winners: Frédéric Blanqui for "Definitions by Rewriting in the Calculus of Constructions," and Kazushige Terui for "Light Affine Lambda Calculus and Polytime Strong Normalization." I'd like to congratulate them both.

The people involved with the conference organization, the program committee, and the (many!) outside reviewers used by the program committee members are all listed on the following pages. I'd like to thank them all; the conference could not have happened without their efforts. I'd like to add a special note of thanks to someone whose name is not listed so prominently: Jon Riecke. Jon kept up the submissions software, housed at Lucent, even after he left Lucent for a startup.

I hope you will find that the contents of these Proceedings were worth the effort required to create them.

Joe Halpern

LICS 2001 Program Chair

Conference Organization

Conference Chair

Harry Mairson, *Boston University*

Publicity Chair

Martin Grohe, *U. Illinois, Chicago*

General Chair

Samson Abramsky, *Oxford U.*

Organizing Committee

Martín Abadi
Samson Abramsky (chair)
Alok Aggarwal
Marc Bezem
Edmund Clarke
Robert Constable
Nachum Dershowitz
Josep Diaz
Harald Ganzinger
Fausto Giunchiglia
Martin Grohe
Daniel Leivant
Leonid Libkin
Giuseppe Longo
Donald A. Martin
John Mitchell
Eugenio Moggi
Vaughan Pratt
Jon Riecke
Simona Ronchi della Rocca
Jerzy Tiuryn
Moshe Y. Vardi
Jeffrey Vitter
Glynn Winskel

Program Committee

Parosh Abdulla, *Uppsala U.*
Rance Cleaveland, *SUNY Stony Brook*
Hubert Comon, *CNRS—ENS Cachan*
Thomas Eiter, *T.U. Vienna*
Erich Grädel, *RWTH Aachen*
Joseph Halpern, *Cornell U. (chair)*
Nevin Heintze, *Bell Labs*
Radha Jagadeesan, *Loyola U.*
Jean-Pierre Jouannaud, *U. Paris-Sud*
Patrick Lincoln, *SRI International*
David McAllester, *AT&T Labs*
Ron van der Meyden, *U. New South Wales*
Adolfo Piperno, *U. Roma "La Sapienza"*
Gordon Plotkin, *U. Edinburgh*
Michel de Rougemont, *U. Paris-II*
Thomas Streicher, *T.U. Darmstadt*
Paweł Urzyczyn, *U. Warsaw*
Pierre Wolper, *U. Liege*

LICS Advisory Board

Martín Abadi
Serge Abiteboul
Samson Abramsky
Mariangiola Dezani
Joseph Halpern
Russell Impagliazzo
Dexter Kozen
John Mitchell
Leszek Pacholski
Andre Scedrov
Dana Scott
Jeanette Wing

Additional Reviewers

Foto Afrati
Thorsten Altenkirch
Mathias Baaz
Arnold Beckmann
Franco Barbanera
Stefano Berardi
Dietmar Berwanger
Frédéric Blanqui
Achim Blumensath
Michele Boreale
Ahmed Bouajjani
Anna Bucalo
Antonio Bucciarelli
Olivier Carton
Didier Caucal
Patrick Cegielski
Pietro Cenciarelli
Juliusz Chroboczek
Christopher Colby
Evelyne Contejean
Bruno Courcelle
Mats Dam
Vincent Danos
Olivier Danvy
Stephane Demri
Rocco De Nicola
Mariangiola Dezani-Ciancaglini
Dan Dougherty
Arnaud Durand
Nancy Durgin
Uwe Egly
Kai Engelhardt
Chris Fermüller
François Fages
Maribel Fernandez
Andrzej Filinski
Jean-Christophe Filliatre
Alain Finkel
Harald Ganzinger
Philippa Gardner
Simon Gay
Andreas Goerdt
Georg Gottlob
Bernhard Gramlich
Herman Geuvers
Etienne Grandjean
Martin Grohe
David Gross
Stefano Guerrini
Bruno Guillaume
Vineet Gupta
Hugo Herbelin
Miki Hermann
Colin Hirsch
Martin Hofmann
Furio Honsell
Martin Hyland
Benedetto Intrigila
S. Purushothaman Iyer
David Janin
Bengt Jonsson
Fairouz Kamareddine
Alex Kurz
Alan Jeffrey
Marcin Jurdziński
Claude Kirchner
Jan Krajiček
Daniel Krob
Stephan Kreutzer
Anna Labella
Yves Lafont
Jens Lagergren
Yassine Lakhnech
Richard Lassaigne
Jean-Marie Lebars
Ugo de'Liguoro
John Longley
Thomas Lukasiewicz
Ian Mackie
Frederic Magniez
Janos Makowsky
Pasquale Malacaria
Luc Maranget
Jean-Yves Marion
Michel Mauny
John Mitchell
Gopalan Nadathur
Damian Niwinski
Bengt Nordstroem
Sven-Olof Nyström
Ichiro Ogata
Luke Ong
Jaap van Oosten
Martin Otto
Sam Owre
Catuscia Palamidessi
Prakash Panangaden
Francesco Parisi-Presicce
Joachim Parrow
Justin Pearson
Paul Pettersson
Sylvain Peyronnet
Andreas Podelski
Randy Pollack
Riccardo Pucella
Grigore Rosu
Simona Ronchi Della Rocca
Paul Ruet
Harald Ruess
Eike Ritter
Giuseppe Rosolini
Kostis Sagonas
Matteo Salina
Ivano Salvo
Andrea Schal
Philippe Schnoebelen
Thomas Schwentick
Géraud Sénizergues
Natarajan Shankar
Anatol Slissenko
Riccardo Silvestri
Julien Stern
Karel Stokkermans
Colin Stirling
Jurgen Stuber
Vanessa Teague
Ashish Tiwari
Hans Tompits
Jerzy Tyszkiewicz
Xavier Urbain
Helmut Veith
Björn Victor
Sergey Vorobyov
Andrei Voronkov
Johannes Waldmann
Victoria Weissman
Benjamin Werner
Thomas Wilke
Mihalis Yannakakis
Wang Yi
Marisa Venturini Zilli

Invited Talk

Probabilistic polynomial-time process calculus and security protocol analysis (short summary)

J. Mitchell*† A. Ramanathan*
Stanford University

A. Scedrov*‡
University of Pennsylvania

V. Teague*
Stanford University

Abstract

We describe properties of a process calculus that has been developed for the purpose of analyzing security protocols. The process calculus is a restricted form of π-calculus, with bounded replication and probabilistic polynomial-time expressions allowed in messages and boolean tests. To avoid problems expressing security in the presence of nondeterminism, messages are scheduled probabilistically instead of nondeterministically. We prove that evaluation may be completed in probabilistic polynomial time and develop properties of a form of asymptotic protocol equivalence that allows security to be speci£ed using observational equivalence, a standard relation from programming language theory that involves quantifying over possible environments that might interact with the protocol. We also relate process equivalence to cryptographic concepts such as pseudo-random number generators and polynomial-time statistical tests.

1 Introduction

A variety of methods are used for analyzing and reasoning about security protocols. The main systematic or formal approaches include specialized logics such as BAN logic [BAN89, DMP01], special-purpose tools designed for cryptographic protocol analysis [KMM94], and theorem proving [Pau97b, Pau97a] and model-checking methods using general purpose tools [Low96, Mea96, MMS97, Ros95, Sch96]. Although these approaches differ in signi£cant ways, all re¤ect the same basic assumptions about the way an adversary may interact with the protocol or attempt to decrypt encrypted messages . In the common model, largely derived from [DY81] and suggestions found in [NS78] (see, e.g., [CDL+99]), a protocol adversary is allowed to nondeterministically choose among possible actions. This is a convenient idealization, intended to give the adversary a chance to £nd an attack if there is one. In the presence of nondeterminism, however, the set of messages an adversary may use to interfere with a protocol must be restricted severely. For example, if the adversary may perform bit manipulation on data, then a nondeterministic adversary may guess any possible secret key. Therefore, the common "Dolev-Yao assumptions" only allow an adversary to construct new messages from indivisible data that are either known from the start or found in messages overheard on the network. Although the Dolev-Yao assumptions make protocol analysis tractable, they also make it possible to "verify" protocols that are in fact susceptible to simple attacks that lie outside the adversary model. Another limitation is that a deterministic or nondeterministic setting does not allow us to analyze probabilistic protocols.

This invited talk will describe some general concepts in security protocol analysis, mention some of the competing approaches, and describe some technical properties of a process calculus that was proposed earlier [LMMS98, LMMS99] as the basis for a form of protocol analysis that is formal, yet closer in foundations to the mathematical setting of modern cryptography. The framework relies on a language for de£ning probabilitic polynomial-time functions [MMS98]. The reason we restrict processes to probabilistic polynomial time is so that we can reason about the security of protocols by quantifying over all "adversarial" processes de£nable in the language. In effect, establishing a bound on the running time of an adversary allows us to relax other simplifying assumptions. Speci£cally, it is possible to consider adversaries that might send randomly chosen messages, or perform sophisticated (yet probabilistic polynomial-time) computation to derive an attack from messages it overhears on the network. A useful aspect of our framework is that we can analyze probabilistic as well as deterministic encryption functions and protocols. With-

*Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, and DARPA Contract N66001-00-C-8015

†Additional support from NSF Grant CCR-9629754.

‡Additional support from NSF Grant CCR-9800785.

0-7695-1281-X/01 $10.00 © 2001 IEEE

out a probabilistic framework, it would not be possible to analyze an encryption function such as ElGamal [ElG85], for which a single plaintext may have more than one ciphertext.

The work has been carried out in collaboration with P. Lincoln, M. Mitchell, A. Scedrov, A. Ramanathan, and V. Teague. The main ideas are outlined in [LMMS98], with the term language presented in [MMS98] and further example protocols considered in [LMMS99]. The closest technical precursor is the Abadi and Gordon spi-calculus [AG99, AG98] which uses observational equivalence and channel abstraction but does not involve probability or computational complexity bounds; subsequent related work is cited in [AF01], for example. Prior work on CSP and security protocols, e.g., [Ros95, Sch96], also uses process calculus and security specifications in the form of equivalence or related approximation orderings on processes.

Although our main long-term objective is to base protocol analysis on standard cryptographic assumptions, this framework may also shed new light on basic questions in cryptography. In particular, the characterization of "secure" encryption function, for use in protocols, does not appear to have been completely settled. While the definition of *semantic security* in [GM84] appears to have been accepted, there are stronger notions such as *non-malleability* [DDN91] that are more appropriate to protocol analysis. In a sense, the difference is that semantic security is natural for the single transmission of an encrypted message, while non-malleability accounts for vulnerabilities that may arise in more complex protocols. Our framework provides a setting for working backwards from security properties of a protocol to derive necessary properties of underlying encryption primitives. While we freely admit that much more needs to be done to produce a systematic analysis method, we believe that a foundational setting for protocol analysis that incorporates probability and complexity restrictions has much to offer in the future.

Slides from this talk will be available on the first author's web site at http://www.stanford.edu/~jcm.

Acknowledgements: Thanks to M. Abadi, D. Boneh, R. Canetti, C. Dwork, R. van Glabbeek, A. Jeffrey, S. Kannan, B. Kapron, P. Lincoln, R. Milner, M. Mitchell, M. Naor, and P. Panangaden for helpful discussions and advice on relevant literature.

References

[AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[AG97] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proc. 4-th ACM Conference on Computer and Communications Security*, pages 36–47, 1997. Revised and expanded versions in *Information and Computation* 148(1999):1-70 and as SRC Research Report 149 (January 1998).

[AG98] M. Abadi and A. Gordon. A bisimulation method for cryptographic protocol. In *Proc. ESOP'98, Springer Lecture Notes in Computer Science*, 1998.

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 143:1–70, 1999. Expanded version available as SRC Research Report 149 (January 1998).

[AR00] M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). In *IFIP International Conference on Theoretical Computer Science*, Sendai, Japan, 2000. Full paper to appear in *J. of Cryptology*.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society, Series A*, 426(1871):233–271, 1989. Also appeared as SRC Research Report 39 and, in a shortened form, in ACM Transactions on Computer Systems 8, 1 (February 1990), 18-36.

[Can00] R. Canetti. A unified framework for analyzing security of protocols. Cryptology ePrint Archive: Report 2000/067; see http://eprint.iacr.org/2000/067/, 2000.

[CDL+99] I. Cervesato, N.A. Durgin, P.D. Lincoln, J.C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12-th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[DDN91] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography (extended abstract). In *Proc. 23rd Annual ACM Symposium on the Theory of Computing*, pages 542–552, 1991.

[DMP01] N.A. Durgin, J.C. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *IEEE Computer Security Foundations Workshop*, page (to appear), 2001.

[DY81] D. Dolev and A. Yao. On the security of public-key protocols. In *Proc. 22nd Annual*

IEEE Symp. Foundations of Computer Science, pages 350–357, 1981.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31:469–472, 1985.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Computer and System Sciences*, 28:281–308, 1984.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

[LMMS98] P.D. Lincoln, M. Mitchell, J.C. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In M.K. Reiter, editor, *Proc. 5-th ACM Conference on Computer and Communications Security*, pages 112–121, San Francisco, California, 1998. ACM Press.

[LMMS99] P.D. Lincoln, J.C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In J.M. Wing and J. Woodcock and J. Davies, editor, *Formal Methods World Congress, Vol. I*, pages 776–793, Toulouse, France, 1999. Springer LNCS 1708.

[Low96] G. Lowe. Breaking and £xing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[Lub96] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton Computer Science Notes, Princeton University Press, 1996.

[Mea96] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: a comparison of two approaches. In *Proc. European Symposium On Research In Computer Security*. Springer Verlag, 1996.

[MMS97] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proc. IEEE Symp. Security and Privacy*, pages 141–151, 1997.

[MMS98] J.C. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proc. 39-th Annual IEEE Symposium on Foundations of Computer Science*, pages 725–733, Palo Alto, California, 1998. IEEE Computer Society Press.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–9, 1978.

[Pau97a] L.C. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th IEEE Computer Security Foundations Workshop*, pages 84–95, 1997.

[Pau97b] L.C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.

[PW00] B. P£tzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *7-th ACM Conference on Computer and Communications Security, Athens, November 2000*, pages 245–254. ACM Press, 2000. Preliminary version: IBM Research Report RZ 3234 (# 93280) 06/12/00, IBM Research Division, Zürich, June 2000.

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW VIII*, page 98. IEEE Computer Soc Press, 1995.

[Sch96] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[Yao82] A. Yao. Theory and applications of trapdoor functions. In *IEEE Foundations of Computer Science*, pages 80–91, 1982.

Session 1

Definitions by Rewriting in the Calculus of Constructions

Frédéric Blanqui

LRI, bât. 490, Université Paris-Sud, 91405 Orsay Cedex, France

tel: +33 (0) 1 69 15 42 35 fax: +33 (0) 1 69 15 65 86

blanqui@lri.fr

Abstract : *The main novelty of this paper is to consider an extension of the Calculus of Constructions where predicates can be defined with a general form of rewrite rules.*

We prove the strong normalization of the reduction relation generated by the β-rule and the user-defined rules under some general syntactic conditions including confluence.

As examples, we show that two important systems satisfy these conditions : a sub-system of the Calculus of Inductive Constructions which is the basis of the proof assistant Coq, and the Natural Deduction Modulo a large class of equational theories.

1 Introduction

This work aims at defining an expressive language allowing to specify and prove mathematical properties in which functions and predicates can be defined by rewrite rules, hence enabling the automatic proof of equational problems.

The Calculus of Constructions. The quest for such a language started with Girard's system F [19] on one hand and De Bruijn's Automath project [18] on the other hand. Later, Coquand and Huet combined both calculi into the Calculus of Constructions (CC) [10]. As in system F, in CC, data structures are defined by using an impredicative encoding which is difficult to use in practice. Following Martin-Löf's theory of types [24], Coquand and Paulin-Mohring defined an extension of CC with inductive types and their associated induction principles as first-class objects : the Calculus of Inductive Constructions (CIC) [26] which is the basis of the proof-assistant Coq [17].

Reasoning Modulo. Defining functions or predicates by recursion is not always convenient. Moreover, with such definitions, equational reasoning is uneasy and leads to very large proof terms. Yet, for decidable theories, equational proofs need not to be kept in proof terms. This idea that proving is not only reasoning (undecidable) but also computing (decidable) has been recently formalized in a general way by Dowek, Hardin and Kirchner with the Natural Deduction Modulo (NDM) for first-order logic [12].

Object-level rewriting. In CC, the first extension by a general notion of rewriting is the λR-cube of Barbanera, Fernández and Geuvers [1]. Their work extends the works of Breazu-Tannen and Gallier [8] and Jouannaud and Okada [21] on the combination of typed λ-calculi with rewriting. The notion of rewriting considered in [21, 1] is not restricted to first-order rewriting, but also includes higher-order rewriting following Jouannaud and Okada's General Schema [21], a generalization of the primitive recursive definition schema. This schema has been reformulated and enhanced so as to deal with definitions on strictly-positive inductive types [5] and with higher-order pattern-matching [3].

Predicate-level rewriting. The notion of rewriting considered in [1] is restricted to the object-level while, in CIC or NDM, it is possible to define predicates by recursion or by rewriting respectively. Recursion at the predicate-level is called "strong elimination" in [26] and has been shown consistent by Werner [31].

Our contributions. The main contribution of our work is a strong normalization result for the Calculus of Constructions extended with, at the predicate-level, user-defined rewrite rules satisfying some general admissibility conditions. As examples, we show that these conditions are satisfied by a sub-system of CIC with strong elimination [26] and the Natural Deduction Modulo [13] a large class of equational theories.

So, our work can be used as a foundation for an extension of a proof assistant like Coq [17] where users could define functions and predicates by rewrite rules. Checking the admissibility conditions or the convert-

0-7695-1281-X/01 $10.00 © 2001 IEEE

ibility of two expressions may require the use of external specialized tools like CiME [16] or ELAN [15].

Outline of the paper. In Section 2, we introduce the Calculus of Algebraic Constructions and our notations. In Section 3, we present our general syntactic conditions. In Section 4, we apply our result to CIC and NDM. In Section 5, we summarize the main contributions of our work and, in Section 6, we give future directions of work. Detailed proofs can be found in [4].

2 The Calculus of Algebraic Constructions (CAC)

2.1 Syntax and notations

We assume the reader familiar with the basics of rewriting [11] and typed λ-calculus [2].

Sorts and symbols. Throughout the paper, we let $\mathcal{S} = \{\star, \Box\}$ be the set of *sorts* where $\star$ denotes the impredicative universe of propositions and $\Box$ a predicative universe containing $\star$. We also assume given a family $\mathcal{F} = (\mathcal{F}_n^s)_{n\geq 0}^{s\in\mathcal{S}}$ of sets of *symbols* and a family $\mathcal{X} = (\mathcal{X}^s)^{s\in\mathcal{S}}$ of infinite sets of *variables*. A symbol $f \in \mathcal{F}_n^s$ is said to be of *arity* $\alpha_f = n$ and sort s. $\mathcal{F}^s$, $\mathcal{F}_n$, $\mathcal{F}$ and $\mathcal{X}$ respectively denote the set of symbols of sort s, the set of symbols of arity n, the set of all symbols and the set of all variables.

Terms. The *terms* of the corresponding CAC are given by the following syntax :

$$t ::= s \mid x \mid f(\vec{t}) \mid (x:t)t \mid [x:t]t \mid tt$$

where $s \in \mathcal{S}$, $x \in \mathcal{X}$ and f is applied to a vector $\vec{t}$ of n terms if $f \in \mathcal{F}_n$. $[x:U]t$ is the abstraction and $(x:U)V$ is the product. A term is *algebraic* if it is a variable or of the form $f(\vec{t})$ with each t_i algebraic.

Notations. As usual, we consider terms up to α-conversion. We denote by $FV(t)$ the set of free variables of t, by $FV^s(t)$ the set $FV(t) \cap \mathcal{X}^s$, by $t\{x \mapsto u\}$ the term obtained by substituting in t every free occurrence of x by u, by $dom(\theta)$ the domain of the substitution θ, by $dom^s(\theta)$ the set $dom(\theta) \cap \mathcal{X}^s$, by $Pos(t)$ the set of positions in t (words on the alphabet of positive integers), by $t|_p$ the subterm of t at position p, by $t[u]_p$ the term obtained by replacing $t|_p$ by u in t, and by $Pos(f,t)$ and $Pos(x,t)$ the sets of positions in t where f occurs and x freely occurs respectively. As usual, we write $T \to U$ for a product $(x:T)U$ where $x \notin FV(U)$.

Rewriting. We assume given a set $\mathcal{R}$ of *rewrite rules* defining the symbols in $\mathcal{F}$. The rules we consider are pairs $l \to r$ made of two terms l and r such that l is an algebraic term of the form $f(\vec{l})$ and $FV(r) \subseteq FV(l)$. They induce a rewrite relation $\to_{\mathcal{R}}$ on terms defined by $t \to_{\mathcal{R}} t'$ iff there are $p \in Pos(t)$, $l \to r \in \mathcal{R}$ and a substitution σ such that $t|_p = l\sigma$ and $t' = t[r\sigma]_p$ (matching is first-order). So, $\mathcal{R}$ can be seen as a particular case of Combinatory Reduction System (CRS) [23] (translate $[x:T]u$ into $\Lambda(T,[x]u)$ and $(x:T)U$ into $\Pi(T,[x]U)$) for which higher-order pattern-matching is not necessary.

Reduction. The *reduction relation* of the calculus is $\to = \to_{\mathcal{R}} \cup \to_\beta$ where $\to_\beta$ is defined as usual by $[x:T]u\,t \to_\beta u\{x \mapsto t\}$. We denote by $\to^*$ its reflexive and transitive closure, by $\leftrightarrow^*$ its symmetric, reflexive and transitive closure, and by $t \downarrow^* u$ the fact that t and u have a common reduct.

2.2 Typing

Types of symbols. We assume given a function τ which, to each symbol f, associates a term τ_f, called its *type*, of the form $(\vec{x} : \vec{T})U$ with $|\vec{x}| = \alpha_f$. In contrast with our own previous work [5] or the work of Barbanera, Fernández and Geuvers [1], symbols can have polymorphic as well as dependent types, as it is the case in CIC.

Typing. An *environment* Γ is an ordered list of pairs $x_i:T_i$ saying that x_i is of type T_i. The *typing relation* of the calculus, $\vdash$, is defined by the rules of Figure 1 (where $s, s' \in \mathcal{S}$).

An environment is *valid* if there is a term typable in it. The condition $\Gamma \vdash v : V$ in the (symb) rule insures that Γ is valid in the case where $n = 0$.

Substitutions. Given two valid environments Γ and Δ, a substitution θ is a *well-typed substitution* from Γ to Δ, written $\theta : \Gamma \to \Delta$, if, for all $x \in dom(\Gamma)$, $\Delta \vdash x\theta : x\Gamma\theta$, where $x\Gamma$ denotes the type associated to x in Γ. With such a substitution, if $\Gamma \vdash t : T$ then $\Delta \vdash t\theta : T\theta$.

Logical consistency. As usual, the logical consistency of such a system is proved in three steps.

First, we must make sure that the reduction relation is correct w.r.t. the typing relation : if $\Gamma \vdash t : T$ and $t \to t'$ then $\Gamma \vdash t' : T$. This property, called *subject reduction*, is not easy to prove for extensions of CC [31, 1]. In the following subsection, we give sufficient conditions for it.

The second step is to prove that the reduction relation $\to$ is weakly or strongly normalizing, hence that every well-typed term has a normal form. Together with the confluence, this implies the decidability of the

Figure 1: Typing rules

$$\text{(ax)}\quad \frac{}{\vdash \star : \Box}$$

$$\text{(symb)}\quad \frac{f \in \mathcal{F}^s_n,\ \tau_f = (\vec{x} : \vec{T})U,\ \gamma = \{\vec{x} \mapsto \vec{t}\} \quad \vdash \tau_f : s \quad \Gamma \vdash v : V \quad \forall i,\ \Gamma \vdash t_i : T_i\gamma}{\Gamma \vdash f(\vec{t}) : U\gamma}$$

$$\text{(var)}\quad \frac{\Gamma \vdash T : s \quad x \in \mathcal{X}^s \setminus dom(\Gamma)}{\Gamma, x{:}T \vdash x : T}$$

$$\text{(weak)}\quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s \quad x \in \mathcal{X}^s \setminus dom(\Gamma)}{\Gamma, x{:}U \vdash t : T}$$

$$\text{(prod)}\quad \frac{\Gamma \vdash T : s \quad \Gamma, x{:}T \vdash U : s'}{\Gamma \vdash (x{:}T)U : s'}$$

$$\text{(abs)}\quad \frac{\Gamma, x{:}T \vdash u : U \quad \Gamma \vdash (x{:}T)U : s}{\Gamma \vdash [x{:}T]u : (x{:}T)U}$$

$$\text{(app)}\quad \frac{\Gamma \vdash t : (x{:}U)V \quad \Gamma \vdash u : U}{\Gamma \vdash tu : V\{x \mapsto u\}}$$

$$\text{(conv)}\quad \frac{\Gamma \vdash t : T \quad T \downarrow^* T' \quad \Gamma \vdash T' : s'}{\Gamma \vdash t : T'}$$

typing relation which is essential in proof assistants. In this paper, we will study the strong normalization property.

The third step is to make sure that there is no normal proof of $\bot = (P{:}\star)P$ in the empty environment. Indeed, if $\bot$ is provable then any proposition P is provable. We will not address this problem here.

2.3 Subject reduction

Proving subject reduction for $\rightarrow_\beta$ requires the following property [4] :

$$(x{:}U)V \leftrightarrow^* (x{:}U')V' \ \Rightarrow\ U \leftrightarrow^* U' \ \wedge\ V \leftrightarrow^* V'$$

It is easy to see that this property is satisfied when $\rightarrow$ is confluent, an assumption which is part of our admissibility conditions described in the next section.

For $\rightarrow_\mathcal{R}$, the idea present in all previous works is to require that, for each rule $l \rightarrow r$, there is an environment Γ and a type T such that $\Gamma \vdash l : T$ and $\Gamma \vdash r : T$. However, this approach has an important drawback : in presence of dependent or polymorphic types, it leads to non-left-linear rules.

For example, consider the type $list : \star \rightarrow \star$ of polymorphic lists built from $nil : (A{:}\star)list(A)$ and $cons : (A{:}\star)A \rightarrow list(A) \rightarrow list(A)$, and the concatenation function $app : (A{:}\star)list(A) \rightarrow list(A) \rightarrow list(A)$. To fulfill the previous condition, we must define app as follows :

$$\begin{array}{rcl} app(A, nil(A), \ell) & \rightarrow & \ell \\ app(A, cons(A, x, \ell), \ell') & \rightarrow & cons(A, x, app(A, \ell, \ell')) \end{array}$$

This has two important consequences. The first one is that rewriting is slowed down because of numerous equality tests. The second one is that it may become much more difficult to prove the confluence of the rewrite relation and of its combination with $\rightarrow_\beta$.

We are going to see that we can take the following left-linear definition without loosing the subject reduction property :

$$\begin{array}{rcl} app(A, nil(A'), \ell) & \rightarrow & \ell \\ app(A, cons(A', x, \ell), \ell') & \rightarrow & cons(A, x, app(A, \ell, \ell')) \end{array}$$

Let $l = app(A, cons(A', x, \ell), \ell')$, $r = cons(A, x, app(A, \ell, \ell'))$, Γ be an environment and σ a substitution such that $\Gamma \vdash l\sigma : list(A\sigma)$. We must prove that $\Gamma \vdash r\sigma : list(A\sigma)$. For $\Gamma \vdash l\sigma : list(A\sigma)$, we must have a derivation like :

$$\text{(symb)}\ \frac{\text{(conv)}\ \dfrac{\text{(symb)}\ \dfrac{\Gamma \vdash A'\sigma : \star \quad \Gamma \vdash x\sigma : A'\sigma \quad \Gamma \vdash \ell\sigma : list(A'\sigma)}{\Gamma \vdash cons(A'\sigma, x\sigma, \ell\sigma) : list(A'\sigma)} \quad list(A'\sigma) \downarrow^* list(A\sigma) \quad \Gamma \vdash list(A\sigma) : \star}{\Gamma \vdash cons(A'\sigma, x\sigma, \ell\sigma) : list(A\sigma)}}{\Gamma \vdash l\sigma : list(A\sigma)}$$

Therefore, $A'\sigma \downarrow^* A\sigma$ and we can derive $\Gamma \vdash x\sigma : A\sigma$, $\Gamma \vdash \ell\sigma : list(A\sigma)$ and :

$$\text{(symb)}\ \frac{\Gamma \vdash A\sigma : \star \quad \Gamma \vdash \ell\sigma : list(A\sigma) \quad \ell'\sigma : list(A\sigma)}{\Gamma \vdash app(A\sigma, \ell\sigma, \ell'\sigma) : list(A\sigma)}$$

$$\text{(symb)}\ \frac{\Gamma \vdash A\sigma : \star \quad \Gamma \vdash x\sigma : A\sigma}{\Gamma \vdash r\sigma : list(A\sigma)}$$

The point is that, although l is not typable, from any typable instance $l\sigma$ of l, we can deduce that $A'\sigma \downarrow^* A\sigma$. By this way, we come to the following conditions :

Definition 1 (Type-preserving rewrite rule) A rewrite rule $l \rightarrow r$ is *type-preserving* if there is an environment Γ and a substitution ρ such that, if $l = f(\vec{l})$, $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$ then :

(S1) $dom(\rho) \subseteq FV(l) \setminus dom(\Gamma)$,

(S2) $\Gamma \vdash l\rho : U\gamma\rho$,

(S3) $\Gamma \vdash r : U\gamma\rho$,

(S4) for any substitution σ, environment Δ and type T, if $\Delta \vdash l\sigma : T$ then $\sigma : \Gamma \rightarrow \Delta$,

(S5) for any substitution σ, environment Δ and type T, if $\Delta \vdash l\sigma : T$ then, for all $x \in dom(\rho)$, $x\sigma \downarrow^* x\rho\sigma$.

In our example, it suffices to take $\Gamma = A:\star, x:A, \ell: list(A), \ell':list(A)$ and $\rho = \{A' \mapsto A\}$.

One may wonder how to check these conditions. In practice, the symbols are incrementally defined. So, assume that we have a confluent and strongly normalizing CAC built over $\mathcal{F}$ and $\mathcal{R}$ and that we want to add a new symbol g. Then, given Γ and ρ, it is decidable to check (S1) to (S3) in the CAC built over $\mathcal{F} \cup \{g\}$ and $\mathcal{R}$ since this system is confluent and strongly normalizing. In [4], we give a simple condition ensuring (S4) (Γ simply needs to be well chosen). The condition (S5) is the most difficult to check and may require the confluence of $\rightarrow$.

3 Admissibility conditions

3.1 Inductive structure

Until now, we made few assumptions on symbols or rewrite rules. In particular, we have no notion of inductive type. Yet, the structure of inductive types plays a key role in strong normalization proofs [25]. On the other hand, we want rewriting to be as general as possible by allowing matching on defined symbols and equations among constructors. This is why, in the following, we introduce an extended notion of constructor and a notion of inductive structure which generalize usual definitions of inductive types [26]. Note that, in contrast with our previous work [5], we allow inductive types to be polymorphic and dependent, as it is the case in CIC.

Definition 2 (Constructors) For $\mathcal{G} \subseteq \mathcal{F}$, let $\mathcal{R}_\mathcal{G}$ be the set of rules defining the symbols in $\mathcal{G}$, that is, the rules whose left-hand side is headed by a symbol in $\mathcal{G}$. The set of *free symbols* is $\mathcal{CF} = \{f \in \mathcal{F} \mid \mathcal{R}_{\{f\}} = \emptyset\}$. The set of *defined symbols* is $\mathcal{DF} = \mathcal{F} \backslash \mathcal{CF}$. The set of *constructors* of a free predicate symbol C is $\mathcal{C}o(C) = \{f \in \mathcal{F}^\star \mid \tau_f = (\vec{y} : \vec{U})C(\vec{v}) \text{ and } |\vec{y}| = \alpha_f\}$.

The constructors of C not only include the constructors in the usual sense but every defined symbol whose output type is C. For example, the symbols $0 : int$, $s : int \rightarrow int$, $p : int \rightarrow int$, $+ : int \rightarrow int \rightarrow int$ and $\times : int \rightarrow int \rightarrow int$ defined by the rules $s(p(x)) \rightarrow x$, $p(s(x)) \rightarrow x$ and others for $+$ and $\times$ are all constructors of the type *int* of integers.

Definition 3 (Inductive structure) An *inductive structure* is given by :

- a quasi-ordering $\geq_\mathcal{F}$ on $\mathcal{F}$, called *precedence*, whose strict part, $>_\mathcal{F}$, is well-founded,
- for each $C \in \mathcal{CF}^\square$ such that $\tau_C = (\vec{x} : \vec{T})\star$, a set $Ind(C) \subseteq \{i \in \{1, ..., \alpha_C\} \mid x_i \in \mathcal{X}^\square\}$ of *inductive* positions,
- for each constructor c, a set $Acc(c) \subseteq \{1, .., \alpha_c\}$ of *accessible* positions.

The accessible positions allow the user to describe which patterns can be used for defining functions, and the inductive positions allow to describe the arguments on which the free predicate symbols should be monotone. This allows us to generalize the notion of positivity used in CIC.

Definition 4 (Positive and negative positions) The sets of *positive* positions $Pos^+(T)$ and *negative* positions $Pos^-(T)$ of a term T are mutually defined by induction on T as follows :

- $Pos^+(s) = Pos^+(F(\vec{t})) = Pos^+(X) = \{\varepsilon\}$,
- $Pos^-(s) = Pos^-(F(\vec{t})) = Pos^-(X) = \emptyset$,
- $Pos^\delta((x:V)W) = 1.Pos^{-\delta}(V) \cup 2.Pos^\delta(W)$,
- $Pos^\delta([x:V]W) = 1.Pos(V) \cup 2.Pos^\delta(W)$,
- $Pos^\delta(Vu) = 1.Pos^\delta(V) \cup 2.Pos(u)$,
- $Pos^\delta(VU) = 1.Pos^\delta(V)$,
- $Pos^+(C(\vec{t})) = \{\varepsilon\} \cup \bigcup\{i.Pos^+(t_i) \mid i \in Ind(C)\}$,
- $Pos^-(C(\vec{t})) = \bigcup\{i.Pos^-(t_i) \mid i \in Ind(C)\}$,

where $\delta \in \{-, +\}$, $-+ = -$, $-- = +$.

For example, in $(x:A)B$, B occurs positively while A occurs negatively. Now, with the type *list* of polymorphic lists, A occurs positively in $list(A)$ iff $Ind(list) = \{1\}$.

Definition 5 (Admissible inductive structure) An inductive structure is *admissible* if, for all $C \in \mathcal{CF}^\square$ with $\tau_C = (\vec{x} : \vec{T})\star$:

(I1) $\forall i \in Ind(C), v_i \in \mathcal{X}^\square$,

and for all c with $\tau_c = (\vec{y} : \vec{U})C(\vec{v})$ and $j \in Acc(c)$:

(I2) $\forall i \in Ind(C), Pos(v_i, U_j) \subseteq Pos^+(U_j)$,
(I3) $\forall D \in \mathcal{CF}^\square, D =_\mathcal{F} C \Rightarrow Pos(D, U_j) \subseteq Pos^+(U_j)$,
(I4) $\forall D \in \mathcal{CF}^\square, D >_\mathcal{F} C \Rightarrow Pos(D, U_j) = \emptyset$,
(I5) $\forall F \in \mathcal{DF}^\square, Pos(F, U_j) = \emptyset$,
(I6) $\forall X \in FV^\square(U_j), \exists \iota_X \in \{1, ..., \alpha_C\}, v_{\iota_X} = X$.

For example, with the type *list* of polymorphic lists, $Ind(list) = \{1\}$, $Acc(nil) = \{1\}$ and $Acc(cons) = \{1, 2, 3\}$ is an admissible inductive structure. If we add the type $tree : \star$ and the constructor $node : list(tree) \rightarrow tree$ with $Acc(node) = \{1\}$, we still have an admissible structure.

The condition (I6) means that the predicate-arguments of a constructor must be parameters of the

type they define. One can find a similar condition in the work of Walukiewicz [30] (called "$\star$-dependency") and in the work of Stefanova [27] (called "safeness").

On the other hand, there is no such explicit restriction in CIC. But the elimination scheme is typed in such a way that no very interesting function can be defined on a type not satisfying (I6). For example, consider the type of heterogeneous non-empty lists (we use the CIC syntax here) $listh = Ind(X : \star)\{C_1|C_2\}$ where $C_1 = (A : \star)(x : A)X$ and $C_2 = (A : \star)(x : A)$ $X \to X$. The typing rule for the non dependent elimination schema ($\text{Nodep}_{\star,\star}$) is :

$$\frac{\Gamma \vdash \ell : listh \quad \Gamma \vdash Q : \star \quad \forall i, \Gamma \vdash f_i : C_i\{listh, Q\}}{\Gamma \vdash Elim(\ell, Q)\{f_1|f_2\} : Q}$$

where $C_1\{listh, Q\} = (A : \star)(x : A)Q$ and $C_2\{listh, Q\} = (A : \star)(x : A)listh \to Q \to Q$. Since Q, f_1 and f_2 must be typable in Γ, the result of f_1 and f_2 cannot depend on A or on x. This means that it is possible to compute the length of such a list but not to use an element of the list.

Definition 6 (Primitive, basic and strictly positive predicates) A free predicate symbol C is :

- *primitive* if, for all $D =_{\mathcal{F}} C$, for all constructor d of type $\tau_d = (\vec{y} : \vec{U})D(\vec{w})$ and for all $j \in Acc(d)$, U_j is either of the form $E(\vec{t})$ with $E <_{\mathcal{F}} D$ and E basic, or of the form $E(\vec{t})$ with $E =_{\mathcal{F}} D$.
- *basic* if, for all $D =_{\mathcal{F}} C$, for all constructor d of type $\tau_d = (\vec{y} : \vec{U})D(\vec{w})$ and for all $j \in Acc(d)$, if $E =_{\mathcal{F}} D$ occurs in U_j then U_j is of the form $E(\vec{t})$.
- *strictly positive* if, for all $D =_{\mathcal{F}} C$, for all constructor d of type $\tau_d = (\vec{y} : \vec{U})D(\vec{w})$ and for all $j \in Acc(d)$, if $E =_{\mathcal{F}} D$ occurs in U_j then U_j is of the form $(\vec{z} : \vec{V})E(\vec{t})$ and no occurrence of $D' =_{\mathcal{F}} D$ occurs in $\vec{V}$.

For example, the type *list* of polymorphic lists is basic but not primitive. The type *listint* of lists of integers with the constructors $nilint$: $listint$ and $consint : int \to listint \to listint$ is primitive. And the type *ord* of Brouwer's ordinals with the constructors $0 : ord$, $s : ord \to ord$ and $lim : (nat \to ord) \to ord$ is strictly positive.

Although we do not explicitly forbid to have non-strictly positive predicate symbols, the admissibility conditions we are going to describe in the following subsections will not enable us to define functions on such a predicate. The same restriction applies on CIC while the system of Walukiewicz [30] is restricted to basic predicates and the λR-cube [1] or NDM [13] are restricted to primitive and non-dependent predicates. However, in the following, for lack of space, we will restrict our attention to basic predicates.

3.2 General Schema

The constructors of primitive predicates (remember that they include all symbols whose output type is a primitive predicate), defined by usual first-order rules, are easily shown to be strongly normalizing since the combination of first-order rewriting with $\to_\beta$ preserves strong normalization [8].

On the other hand, in the presence of higher-order rules, few techniques are known :

- Van de Pol [28] extended to the higher-order case the use of strictly monotone interpretations . This technique is very powerful but difficult to use in practice and has not been studied yet in type systems richer than the simply-typed λ-calculus.
- Jouannaud and Okada [21] defined a syntactic criterion, the General Schema, which extends primitive recursive definitions. This schema has been reformulated and enhanced to deal with definitions on strictly-positive types [6], to higher-order pattern-matching [3] and to richer type systems with object-level rewriting [1, 5].
- Jouannaud and Rubio [22] extended to the higher-order case the use of Dershowitz's recursive path ordering. The obtained ordering can be seen as a recursive version of the General Schema and has been extended by Walukiewicz [30] to the Calculus of Constructions with object-level rewriting.

Here, we present an extension of the General Schema defined in [5] to deal with type-level rewriting, the main novelty of our paper.

The General Schema is based on Tait and Girard's computability predicate technique [19] for proving the strong normalization of the simply-typed λ-calculus and system F. This technique consists in interpreting each type T by a set $[\![T]\!]$ of strongly normalizable terms, called *computable* , and in proving that $t \in [\![T]\!]$ whenever $\Gamma \vdash t : T$.

The idea of the General Schema is then to define, from a left-hand side of rule $f(\vec{l})$, a set of right-hand sides r that are computable whenever the l_i's are computable. This set is built from the variables of the left-hand side, called *accessible* , that are computable whenever the l_i's are computable, and is then closed by computability-preserving operations.

For the sake of simplicity, two sequences of arguments of a symbol f will be compared in a lexicographic manner. But it is possible to do these comparisons in a multiset manner or with a simple combination of lexicographic and multiset comparisons (see [4] for details).

Definition 7 (Accessibility) A pair $\langle u, U\rangle$ is *accessible* in a pair $\langle t, T\rangle$, written $\langle t, T\rangle \triangleright_1 \langle u, U\rangle$, if $\langle t, T\rangle = \langle c(\vec{u}), C(\vec{v})\gamma\rangle$ and $\langle u, U\rangle = \langle u_j, U_j\gamma\rangle$ with c a constructor of type $\tau_c = (\vec{y} : \vec{U})C(\vec{v})$, $\gamma = \{\vec{y} \mapsto \vec{u}\}$ and $j \in Acc(c)$.

For example, in the definition of *app* previously given, A', x and ℓ are all accessible in $t = cons(A', x, \ell)$: $\langle t, list(A)\rangle \triangleright_1 \langle A', \star\rangle$, $\langle t, list(A)\rangle \triangleright_1 \langle x, A'\rangle$ and $\langle t, list(A)\rangle \triangleright_1 \langle \ell, list(A')\rangle$.

Definition 8 (Derived type) Let t be a term of the form $l\sigma$ with $l = f(\vec{l})$ algebraic, $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$. Let $p \in Pos(l)$ with $p \neq \varepsilon$. The subterm $t|_p$ of t has a *derived type*, $\tau(t, p)$, defined as follows :
- if $p = i$ then $\tau(t, p) = T_i\gamma\sigma$,
- if $p = iq$ and $q \neq \varepsilon$ then $\tau(t, p) = \tau(t_i, q)$.

Definition 9 (Well-formed rule) Let $R = (l \to r, \Gamma, \rho)$ be a rule with $l = f(\vec{l})$, $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$. The rule R is *well-formed* if, for all $x \in dom(\Gamma)$, there is $i \leq a_f$ and $p_x \in Pos(x, l_i)$ such that $\langle l_i, T_i\gamma\rangle \triangleright_1^* \langle x, \tau(l, ip_x)\rangle$ and $\tau(l, ip_x)\rho = x\Gamma$.

Definition 10 (Computable closure) Let $R = (l \to r, \Gamma_0, \rho)$ be a rule with $l = f(\vec{l})$, $\tau_f = (\vec{x} : \vec{T})U$ et $\gamma = \{\vec{x} \mapsto \vec{l}\}$. The order $>$ on the arguments of f is the lexicographic extension of $\triangleright_1^+$. The *computable closure* of R is the relation $\vdash_c$ defined by the rules of Figure 2.

Definition 11 (General Schema) A rule $(f(\vec{l}) \to r, \Gamma, \rho)$ with $\tau_f = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$ satisfies the *General Schema* if it is well-formed and $\Gamma \vdash_c r : U\gamma\rho$.

It is easy to check that the rules for *app* are well-formed and that $\Gamma \vdash_c cons(A, x, app(A, \ell, \ell')) : list(A)$. For example, we show that $\Gamma \vdash_c app(A, \ell, \ell') : list(A)$:

$$\dfrac{\dfrac{\Gamma \vdash_c \star : \Box}{\Gamma \vdash_c A : \star} \quad \dfrac{\dfrac{\cdots}{\Gamma \vdash_c A : \star}}{\Gamma \vdash_c list(A) : \star} \quad \dfrac{\cdots}{\Gamma \vdash_c \ell : list(A)} \quad \dfrac{\cdots}{\Gamma \vdash_c \ell' : list(A)}}{\langle cons(A', x, \ell), list(A)\rangle > \langle \ell, list(A)\rangle}}{\Gamma \vdash_c app(A, \ell, \ell')}$$

3.3 Admissibility conditions

Definition 12 (Rewrite systems) Let $\mathcal{G}$ be a set of symbols. The *rewrite system* $(\mathcal{G}, \mathcal{R}_\mathcal{G})$ is :
- *algebraic* if :
  - $\mathcal{G}$ is made of predicate symbols or of constructors of primitive predicates,
  - all rules of $\mathcal{R}_\mathcal{G}$ have an algebraic right-hand side;
- *non-duplicating* if, for all $l \to r \in \mathcal{R}_\mathcal{G}$, no variable has more occurrences in r than in l;
- *primitive* if, for all rule $l \to r \in \mathcal{R}_\mathcal{G}$, r is of the form $[\vec{x} : \vec{T}]g(\vec{u})\vec{v}$ with g belonging to $\mathcal{G}$ or g being a primitive predicate symbol;
- *simple* if, for all $g(\vec{l}) \to r \in \mathcal{R}_\mathcal{G}$:
  - all the symbols occuring in $\vec{l}$ are free,
  - for all sequence of terms $\vec{t}$, at most one rule can apply at the top of $g(\vec{t})$,
  - for all rule $g(\vec{l}) \to r \in \mathcal{R}_\mathcal{G}$ and all $Y \in FV^\Box(r)$, there is a unique κ_Y such that $l_{\kappa_Y} = Y$;
- *positive* if, for all $l \to r \in \mathcal{R}_\mathcal{G}$ and all $g \in \mathcal{G}$, $Pos(g, r) \subseteq Pos^+(r)$;

Figure 2: Computable closure

$$\text{(acc)} \quad \dfrac{\Gamma_0 \vdash_c x\Gamma_0 : s \quad x \in dom^s(\Gamma_0)}{\Gamma_0 \vdash_c x : x\Gamma_0}$$

$$\text{(ax)} \quad \dfrac{}{\Gamma_0 \vdash_c \star : \Box}$$

$$\text{(symb}^<\text{)} \quad \dfrac{\begin{array}{c} g \in \mathcal{F}_n^s,\ \tau_g = (\vec{y} : \vec{U})V,\ \gamma = \{\vec{y} \mapsto \vec{u}\} \\ g <_\mathcal{F} f \quad \Gamma \vdash_c \tau_g : s \quad \forall i,\ \Gamma \vdash_c u_i : U_i\gamma \end{array}}{\Gamma \vdash_c g(\vec{u}) : V\gamma}$$

$$\text{(symb}^=\text{)} \quad \dfrac{\begin{array}{c} g \in \mathcal{F}_n^s,\ \tau_g = (\vec{y} : \vec{U})V,\ \gamma = \{\vec{y} \mapsto \vec{u}\} \\ g =_\mathcal{F} f \quad \Gamma \vdash_c \tau_g : s \quad \forall i,\ \Gamma \vdash_c u_i : U_i\gamma \\ \langle \vec{l}, \vec{T}\gamma_0\rangle > \langle \vec{u}, \vec{U}\gamma\rangle \end{array}}{\Gamma \vdash_c g(\vec{u}) : V\gamma}$$

$$\text{(var)} \quad \dfrac{\Gamma \vdash_c T : s \quad x \in \mathcal{X}^s \setminus FV(l)}{\Gamma, x{:}T \vdash_c x : T}$$

$$\text{(weak)} \quad \dfrac{\Gamma \vdash_c t : T \quad \Gamma \vdash_c U : s \quad x \in \mathcal{X}^s \setminus FV(l)}{\Gamma, x{:}U \vdash_c t : T}$$

$$\text{(prod)} \quad \dfrac{\Gamma \vdash_c T : s \quad \Gamma, x{:}T \vdash_c U : s'}{\Gamma \vdash_c (x{:}T)U : s'}$$

$$\text{(abs)} \quad \dfrac{\Gamma, x{:}T \vdash_c u : U \quad \Gamma \vdash_c (x{:}T)U : s}{\Gamma \vdash_c [x{:}T]u : (x{:}T)U}$$

$$\text{(app)} \quad \dfrac{\Gamma \vdash_c t : (x{:}U)V \quad \Gamma \vdash_c u : U}{\Gamma \vdash_c tu : V\{x \mapsto u\}}$$

$$\text{(conv)} \quad \dfrac{\Gamma \vdash_c t : T \quad T \downarrow^* T' \quad \Gamma \vdash_c T' : s'}{\Gamma \vdash_c t : T'}$$

- *recursive* if all the rules of $\mathcal{R}_\mathcal{G}$ satisfy the General Schema;
- *safe* if, for all $(g(\vec{l}) \to r, \Gamma, \rho) \in \mathcal{R}_\mathcal{G}$ with $\tau_g = (\vec{x} : \vec{T})U$ and $\gamma = \{\vec{x} \mapsto \vec{l}\}$:
  - for all $X \in FV^{\square}(\vec{T}U)$, $X\gamma\rho \in dom^{\square}(\Gamma)$,
  - for all $X, X' \in FV^{\square}(\vec{T}U)$, $X\gamma\rho = X'\gamma\rho \Rightarrow X = X'$.

Definition 13 (Admissible CAC) A CAC is *admissible* if :

(A1) $\to = \to_\mathcal{R} \cup \to_\beta$ is confluent;
(A2) its inductive structure is admissible;
(A3) $(\mathcal{DF}^{\square}, \mathcal{R}_{\mathcal{DF}^{\square}})$ is either :
  - primitive,
  - simple and positive,
  - simple and recursive;

(A4) there is a partition $\mathcal{F}_a \uplus \mathcal{F}_{na}$ of $\mathcal{DF}$ (*algebraic* and *non-algebraic* symbols) such that :
  - $(\mathcal{F}_a, \mathcal{R}_{\mathcal{F}_a})$ is algebraic, non-duplicating and strongly normalizing,
  - no symbol of $\mathcal{F}_{na}$ occurs in the rules of $\mathcal{R}_{\mathcal{F}_a}$,
  - $(\mathcal{F}_{na}, \mathcal{R}_{\mathcal{F}_{na}})$ is safe and recursive.

The simplicity condition in (A3) extends to the case of rewriting the restriction in CIC of strong elimination to "small" inductive types, that is, to the types whose constructors have no predicate-arguments except the parameters of the type.

The safeness condition in (A4) means that one cannot do pattern-matching or equality tests on predicate-arguments that are necessary for typing other arguments. In her extension of HORPO to the Calculus of Constructions, Walukiewicz requires similar conditions [30].

The non-duplication condition in (A4) ensures the modularity of the strong normalization. Indeed, in general, the combination of two strongly normalizing rewrite systems is not strongly normalizing.

Now, for proving (A1), one can use the following result of van Oostrom [29] (remember that $\mathcal{R} \cup \beta$ can be seen as a CRS [23]) : the combination of two confluent left-linear CRS's having no critical pairs between each other is confluent. So, since $\to_\beta$ is confluent and $\mathcal{R}$ and β cannot have critical pairs between each other, if $\mathcal{R}$ is left-linear and confluent then $\to_\mathcal{R} \cup \to_\beta$ is confluent. Therefore, our conditions (S1) to (S5) are very useful to eliminate the non-linearities due to typing reasons.

We can now state our main result. You can find a detailed proof in [4].

Theorem 14 (Strong normalization) Any admissible CAC is strongly normalizing.

The proof is based on Coquand and Gallier's extension to the Calculus of Constructions [9] of Tait and Girard's computability predicate technique [19]. As explained before, the idea is to define an interpretation for each type and to prove that each well-typed term belongs to the interpretation of its type.

The main difficulty is to define an interpretation for predicate symbols that is invariant by reduction, a condition required by the type conversion rule (conv).

Thanks to the positivity conditions, the interpretation of a free predicate symbol can be defined as the least fixpoint of a monotone function over the lattice of computability predicates.

For the defined predicate symbols, it depends on the kind of system $(\mathcal{DF}^{\square}, \mathcal{R}_{\mathcal{DF}^{\square}})$ is. If it is primitive then we simply interpret it as the set of strongly normalizable terms. If it is positive then, thanks to the positivity condition, we can interpret it as a least fixpoint. Finally, if it is recursive then we can define its interpretation recursively, the General Schema providing a well-founded definition.

4 Examples

4.1 Calculus of Inductive Constructions

We are going to see that we can apply our strong normalization theorem to a sub-system of CIC [26] by translating it into an admissible CAC. The first complete proof of strong normalization of CIC (with strong elimination) is due to Werner [31] who, in addition, considers η-reductions in the type conversion rule.

In CIC, one has strictly-positive inductive types and the corresponding induction principles. We recall the syntax and the typing rules of CIC but, for the sake of simplicity, we will restrict our attention to basic inductive types and non-dependent elimination schemas. For a complete presentation, see [4].

- Inductive types are denoted by $Ind(X : A)\{\vec{C}\}$ where the C_i's are the types of the constructors. The term A must be of the form $(\vec{x} : \vec{A})\star$ and the C_i's of the form $(\vec{z} : \vec{B})X\vec{m}$.
- The i-th constructor of an inductive type I is denoted by $Constr(i, I)$.
- Recursors are denoted by $Elim(I, Q, \vec{a}, c)$ where I is the inductive type, Q the type of the result, $\vec{a}$ the arguments of I and c a term of type $I\vec{a}$.

The typing rules for these constructions are given in Figure 3. The rules for the other constructions are the same as for the Calculus of Constructions.

If $C_i = (\vec{z} : \vec{B})X\vec{m}$ then $C_i\{I, Q\}$ denotes $(\vec{z} : \vec{B})(\vec{z}' : \vec{B}\{X \mapsto Q\})\, Q\vec{m}$. The reduction relation associated to

Figure 3: Typing rules of CIC

$$(\text{Ind}_\star)\quad \frac{\forall i,\ \Gamma, X:A \vdash C_i : \star}{\Gamma \vdash Ind(X:A)\{\vec{C}\} : A}$$

$$(\text{Constr})\ \frac{\Gamma \vdash I = Ind(X:A)\{\vec{C}\} : A}{\Gamma \vdash Constr(i,I) : C_i\{X \mapsto I\}}$$

$$(\text{Nodep}_{\star,s})\ \frac{\Gamma \vdash c : I\vec{a} \quad \Gamma \vdash Q : (\vec{x}:\vec{A})s \quad \forall i,\ \Gamma \vdash f_i : C_i\{I,Q\}}{\Gamma \vdash Elim(I,Q,\vec{a},c)\{\vec{f}\} : Q\vec{a}}$$

Elim is called *ι-reduction* and is defined as follows :

$$Elim(I,Q,\vec{a},Constr(i,I')\,\vec{b})\{\vec{f}\} \to_\iota f_i\,\vec{b}\,\vec{b'}$$

where, if $C_i = (\vec{z}:\vec{B})X\vec{m}$, then $b'_j = Elim(I,Q,\vec{a}',b_j)$ if $B_j = X\vec{a}'$, and $b'_j = b_j$ otherwise.

Now, we consider the sub-system CIC$^-$ obtained by applying the following restrictions :

- In the typing rules ($\text{Ind}_\star$) and (Constr), we assume that Γ is empty since, in CAC, the types of the symbols must be typable in the empty environment.
- In the rule ($\text{Nodep}_{\star,\star}$) (the one for weak elimination), we require Q to be typable in the empty environment.
- In the rule ($\text{Nodep}_{\star,\Box}$) (the one for strong elimination), instead of requiring $\Gamma \vdash Q : (\vec{x}:\vec{A})\Box$ which is not possible in the Calculus of Constructions since $\Box$ is not typable, we require Q to be a closed term of the form $[\vec{x}:\vec{A}]K$ with K of the form $(\vec{y}:\vec{U})\star$.
- We assume that every inductive type satisfies (I6).

Theorem 15 CIC$^-$ can be translated into an admissible CAC, hence is strongly normalizing.

We define the translation $\langle\ \rangle$ by induction on the size of terms :

- Let $I = Ind(X:A)\{\vec{C}\}$. We define $\langle I\rangle = [\vec{x}:\langle\vec{A}\rangle]\ Ind_I(\vec{x})$ where Ind_I is a symbol of type $(\vec{x}:\langle\vec{A}\rangle)\star$.
- By assumption, $C_i = (\vec{z}:\vec{B})X\vec{m}$. We define $\langle Constr(i,I)\rangle = [\vec{z}:\vec{B}]Constr^i_I(\vec{z})$ where $Constr^i_I$ is a symbol of type $(\vec{z}:\langle\vec{B}\rangle)Ind_I(\langle\vec{m}\rangle)$.
- Let $T_i = C_i\{I,Q\}$. If $Q = [\vec{x}:\vec{A}]K$ then we define $\langle Elim(I,Q,\vec{a},c)\{\vec{f}\}\rangle = SElim^Q_I(\langle\vec{f}\rangle,\langle\vec{a}\rangle,\langle c\rangle)$ where $SElim^Q_I$ is a symbol of type $(\vec{f}:\langle\vec{T}\rangle)\ (\vec{x}:\langle\vec{A}\rangle)\ \langle K\rangle$. Otherwise, we define $\langle Elim(I,Q,\vec{a},c)\{\vec{f}\}\rangle = WElim_I(\langle Q\rangle,\langle\vec{f}\rangle,\langle\vec{a}\rangle,\langle c\rangle)$ where $WElim_I$ is a symbol of type $(Q:\langle A\rangle)(\vec{f}:\langle\vec{T}\rangle)(\vec{x}:\langle\vec{A}\rangle)\ \langle Q\rangle\vec{x}$.
- The other terms are defined recursively ($\langle uv\rangle = \langle u\rangle\langle v\rangle,\ldots$).

The ι-reduction is translated by the following rules :

$$SElim^Q_I(\vec{f},\vec{a},Constr^i_I(\vec{b})) \to f_i\,\vec{b}\,\vec{b'}$$
$$WElim_I(Q,\vec{f},\vec{a},Constr^i_I(\vec{b})) \to f_i\,\vec{b}\,\vec{b'}$$

where, if $C_i = (\vec{z}:\vec{B})X\vec{m}$, then $b'_j = SElim^Q_I(\vec{f},\vec{a}',b_j)$ (or $WElim_I(Q,\vec{f},\vec{a}',b_j)$) if $B_j = X\vec{a}'$, and $b'_j = b_j$ otherwise.

Now, we are left to check the admissibility :

(A1) $\to_{\beta\iota}$ is orthogonal, hence confluent [29].

(A2) The inductive structure defined by $I <_{\mathcal{F}} J$ if I is a subterm of J, $Ind(Ind_I) = \emptyset$, $Acc(Constr^i_I) = \{1,..,|\vec{z}|\}$ if $C_i = (\vec{z}:\vec{B})X\vec{m}$, is admissible.

(A3) The rules defining the strong recursors form a simple (they are defined by case on each constructor and only for small inductive types) and recursive rewrite system (they satisfy the General Schema).

(A4) The rules defining the recursors form a safe (except for the constructor, all the arguments are distinct variables) and recursive rewrite system (they satisfy the General Schema).

4.2 Natural Deduction Modulo

NDM for first-order logic [12] can be presented as an extension of Natural Deduction with the additional inference rule :

$$\frac{\Gamma \vdash P}{\Gamma \vdash Q}\quad \text{if } P \equiv Q$$

where $\equiv$ is a congruence relation on propositions. This is a powerful extension of first-order logic since both higher-order logic and set theory with a comprehension symbol can be described in this framework (by using explicit substitutions).

In [13], Dowek and Werner study the termination of cut-elimination in the case where $\equiv$ is induced by a confluent and weakly-normalizing rewrite system. In particular, they prove the termination in two general cases : when the rewrite system is positive and when it is quantifier-free. In [14], they provide an example of confluent and weakly normalizing rewrite system for which cut-elimination is not terminating. The problem comes from the fact that the elimination rule for $\forall$ introduces a substitution :

$$\frac{\Gamma \vdash \forall x.P(x)}{\Gamma \vdash P(t)}$$

Thus, when a predicate symbol is defined by a rule whose right-hand side contains quantifiers, its combi-

nation with β may not preserve normalization. Therefore, a criterion for higher-order rewriting is needed.

Since NDM is a CAC (we can define the logical connectors as inductive types), we can compare in more details the conditions of [13] with our conditions.

(A1) In [13], only $\rightarrow_{\mathcal{R}}$ is required to be confluent. In general, this is not sufficient for having the confluence of $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$. However, if $\mathcal{R}$ is left-linear then $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$ is confluent [29].

(A2) NDM types are primitive and form an admissible inductive structure if we take them equivalent in the relation $\leq_{\mathcal{F}}$.

(A3) In [13], the termination of cut-elimination is proved in two general cases : when $(\mathcal{DF}^{\Box}, \mathcal{R}_{\mathcal{DF}^{\Box}})$ is quantifier-free and when it is positive. Quantifier-free rewrite systems are primitive. So, in this case, (A3) is satisfied. In the positive case, we require that left-hand sides are made of free symbols and that at most one rule can apply at the top of a term. On the other hand, we provide a new case : $(\mathcal{DF}^{\Box}, \mathcal{R}_{\mathcal{DF}^{\Box}})$ can be simple and recursive.

(A4) Quantifier-free rules are algebraic and rules with quantifiers are not. In [13], these two kinds of rules are treated in the same way but the counter-example given in [14] shows that they should not. In CAC, we require that the rules with quantifiers satisfy the General Schema.

Theorem 16 A NDM system satisfying (A1), (A3) and (A4) is admissible, hence strongly normalizing.

4.3 CIC + Rewriting

As a combination of the two previous applications, our work shows that the extension of CIC$^-$ with user-defined rewrite rules, even at the predicate-level, is sound if these rules follow our admissibility conditions.

As an example, we consider simplification rules on propositions that are not definable in CIC. Assume that we have the symbols $\vee : \star \rightarrow \star \rightarrow \star$, $\wedge : \star \rightarrow \star \rightarrow \star$, $\neg : \star \rightarrow \star$, $\bot : \star$, $\top : \star$, and the rules :

$$\begin{array}{lll} \top \vee P \rightarrow \top & \bot \wedge P \rightarrow \bot & \neg\top \rightarrow \bot \\ P \vee \top \rightarrow \top & P \wedge \bot \rightarrow \bot & \neg\bot \rightarrow \top \end{array}$$

$$\neg(P \wedge Q) \rightarrow \neg P \vee \neg Q \qquad \neg(P \vee Q) \rightarrow \neg P \wedge \neg Q$$

The predicate constructors $\vee$, $\wedge$, ... are all primitive. The rewrite system is primitive, algebraic, strongly normalizing and confluent (this can be automatically proved by CiME [16]). Since it is left-linear, its combination with $\rightarrow_{\beta}$ is confluent [29]. Therefore, it is an admissible CAC. But it lacks many other rules [20] which requires rewriting modulo associativity and commutativity, an extension we leave for future work.

5 Conclusion

We have defined an extension of the Calculus of Constructions by functions and predicates defined with rewrite rules. The main contributions of our work are the following :

- We consider a general notion of rewriting at the predicate-level which generalizes the "strong elimination" of the Calculus of Inductive Constructions [26, 31]. For example, we can define simplification rules on propositions that are not definable in CIC.
- We consider general syntactic conditions, including confluence, that ensure the strong normalization of the calculus. In particular, these conditions are fulfilled by two important systems : a sub-system of the Calculus of Inductive Constructions which is the basis of the proof assistant Coq [17], and the Natural Deduction Modulo [12, 13] a large class of equational theories.
- We use a more general notion of constructor which allows pattern-matching on defined symbols and equations among constructors.
- We relax the usual conditions on rewrite rules for ensuring the subject reduction property. By this way, we can eliminate some non-linearities in left-hand sides of rules and ease the confluence proof.

6 Directions for future work

- In our conditions, we assume that the predicate symbols defined by rewrite rules containing quantifiers ("non-primitive" predicate symbols) are defined by pattern-matching on free symbols only ("simple" systems). It would be nice to be able to relax this condition.
- Another important assumption is that the reduction relation $\rightarrow = \rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$ must be confluent. We will try to find sufficient conditions on $\mathcal{R}$ in order to get the confluence of $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$. In the simply-typed λ-calculus, if $\mathcal{R}$ is a first-order rewrite system then the confluence of $\mathcal{R}$ is a sufficient condition [7]. But few results are known in the case of a richer type system or of higher-order rewriting.
- Finally, we expect to extend this work with rewriting modulo some useful equational theories like associativity and commutativity, and also by allowing η-reductions in the type conversion rule.

Acknowledgments : I would like to thank Daria Walukiewicz, Gilles Dowek, Jean-Pierre Jouannaud and Christine Paulin for useful comments on previous versions of this work.

References

[1] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

[2] H. Barendregt. Lambda calculi with types. In S. Abramski, D. Gabbay, and T. Maibaum, editors, *Handbook of logic in computer science*, volume 2. Oxford University Press, 1992.

[3] F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. of RTA'00*, LNCS 1833.

[4] F. Blanqui. *Théorie des Types et Réécriture (Type Theory and Rewriting)*. PhD thesis, Université Paris-Sud (France), 2001. Available at `http://www.lri.fr/~blanqui`. An english version will be available soon.

[5] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *Proc. of RTA'99*, LNCS 1631.

[6] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 277, 2001.

[7] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of LICS'88*. IEEE Computer Society.

[8] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, 1991.

[9] T. Coquand and J. Gallier. A proof of strong normalization for the Theory of Constructions using a Kripke-like interpretation, 1990. Paper presented at the 1st Int. Work. on Logical Frameworks but not published in the proceedings. Available at `ftp://ftp.cis.upenn.edu/pub/papers/gallier/sntoc.dvi.Z`.

[10] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[11] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. North-Holland, 1990.

[12] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Technical Report 3400, INRIA Rocquencourt (France), 1998.

[13] G. Dowek and B. Werner. Proof normalization modulo. In *Proc. of TYPES'98*, LNCS 1657.

[14] G. Dowek and B. Werner. An inconsistent theory modulo defined by a confluent and terminating rewrite system, 2000. Available at `http://pauillac.inria.fr/~dowek/`.

[15] C. Kirchner *et al.* ELAN, 2000. Available at `http://elan.loria.fr/`.

[16] C. Marché *et al.* CiME, 2000. Available at `http://www.lri.fr/~demons/cime.html`.

[17] C. Paulin *et al.* *The Coq Proof Assistant Reference Manual Version 6.3.1*. INRIA Rocquencourt (France), 2000. Available at `http://coq.inria.fr/`.

[18] H. Geuvers, R. Nederpelt, and R. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[19] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[20] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25:255–300, 1985.

[21] J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[22] J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proc. of LICS'99*, IEEE Computer Society.

[23] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems : introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[24] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Napoli, Italy, 1984.

[25] N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, United States, 1987.

[26] C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *Proc. of TLCA'93*, LNCS 664.

[27] M. Stefanova. *Properties of Typing Systems*. PhD thesis, Nijmegen University (Netherlands), 1998.

[28] J. van de Pol. *Termination of higher-order rewrite systems*. PhD thesis, University of Utrecht, Nederlands, 1994.

[29] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Netherlands, 1994.

[30] D. Walukiewicz. Termination of rewriting in the Calculus of Constructions. In *Proc. of LFM'00*.

[31] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, France, 1994.

Deconstructing Shostak*

Harald Rueß and Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{ruess,shankar}@csl.sri.com
Phone: (650)859-5272
Fax: (650)859-2844

Abstract

Decision procedures for equality in a combination of theories are at the core of a number of verification systems. Shostak's decision procedure for equality in the combination of solvable and canonizable theories has been around for nearly two decades. Variations of this decision procedure have been implemented in a number of systems including STP, EHDM, *PVS, STeP, and SVC. The algorithm is quite subtle and a correctness argument for it has remained elusive. Shostak's algorithm and all previously published variants of it yield incomplete decision procedures. We describe a variant of Shostak's algorithm along with proofs of termination, soundness, and completeness.*

1 Introduction

In 1984, Shostak [Sho84] published a decision procedure for the quantifier-free theory of equality over uninterpreted functions combined with other theories that are canonizable and solvable. Such algorithms decide statements of the form $T \vdash a = b$, where T is a collection of equalities, and T, a, and b contain a mixture of interpreted and uninterpreted function symbols. This class of statements includes a large fraction of the proof obligations that arise in verification including those involving extended typechecking, verification conditions generated from Hoare triples, and inductive theorem proving. Shostak's procedure is at the core of several verification systems including STP [SSMS82], EHDM [EHD93], PVS [ORS92], STeP [MT96, Bjø99], and SVC [BDL96]. The soundness of Shostak's algorithm is reasonably straightforward, but its completeness has steadfastly resisted proof. The proof given by Shostak [Sho84] is seriously flawed. Despite its significance and popularity, Shostak's original algorithm and its subsequent variations [CLS96, BDL96, Bjø99] are all incomplete and potentially nonterminating. We explain the ideas underlying Shostak's decision procedure by presenting a correct version of the algorithm along with rigorous proofs for its correctness.

If the terms in a conjecture of the form $T \vdash a = b$ are constructed solely from variables and uninterpreted function symbols, then congruence closure [NO80, Sho78, DST80, CLS96, Kap97, BRRT99] can be used to partition the subterms into equivalence classes respecting T and congruence. For example, when congruence closure is applied to

$$f^3(x) = f(x) \vdash f^5(x) = f(x),$$

the equivalence classes generated by the antecedent equality are $\{x\}, \{f(x), f^3(x), f^5(x)\}$, and $\{f^2(x), f^4(x)\}$. This partition clearly validates the conclusion $f^5(x) = f(x)$.

In practice, a conjecture $T \vdash a = b$ usually contains a mixture of uninterpreted and interpreted function symbols. Semantically, uninterpreted functions are unconstrained, whereas interpreted function are constrained by a *theory*, i.e., a closure condition with respect to consequence on a set of equalities. An example of such an assertion is

$$f(x-1)-1 = x+1,\ f(y)+1 = y-1,\ y+1 = x \vdash \mathit{false},$$

where $+$, $-$, and the numerals are from the theory of linear arithmetic, *false* is an abbreviation for $0 = 1$, and f is an uninterpreted function symbol. The contradiction here cannot be derived solely by congruence closure or linear arithmetic. Linear arithmetic is used to show that $x - 1 = y$ so that $f(x-1) = f(y)$ follows by congruence. Linear arithmetic can then be used to show that $x + 2 = y - 2$ which contradicts $y + 1 = x$.

*This work was supported by SRI International, and by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-0079.

0-7695-1281-X/01 $10.00 © 2001 IEEE

Nelson and Oppen [NO79] showed how decision procedures for disjoint equational theories could be combined. Since linear arithmetic and uninterpreted equality are disjoint, this method can be applied to the above example. First, *variable abstraction* is used to obtain a theory-wise partition of the *term universe*, i.e., the subterms of T, a, and b, in a conjecture $T \vdash a = b$. The uninterpreted equality theory Q then consists of the terms $\{f(u), f(y), w, z\}$ and the equalities $\{w = f(u), z = f(y)\}$, and the linear arithmetic theory L consists of the terms $\{u, x, y, x - 1, w - 1, x + 1, z + 1, y - 1, y + 1\}$ and the equalities $\{u = x - 1, w - 1 = x + 1, z + 1 = y - 1, y + 1 = x\}$. The key observation is that once the terms and equalities have been partitioned using variable abstraction, the two theories L and Q need exchange only equalities between variables. Thus, linear arithmetic can be used to derive the equality $u = y$, from which congruence closure derives $w = z$, and the contradiction then follows from linear arithmetic. Since the term universe is fixed in advance, there are only a bounded number of equalities between variables so that the propagation of information between the decision procedures must ultimately converge.

The Nelson-Oppen combination procedure has some disadvantages. The individual decision procedures must carry out their own equality propagation and the communication of equalities between decision procedures can be expensive. The number of equalities is quadratic in the size of the term universe, and each closure operation can itself be linear in the size of the term universe.

Shostak's algorithm tries to gain efficiency by maintaining and propagating equalities within a single congruence closure data structure. Equalities involving interpreted symbols contain more information than uninterpreted equalities. For example, the equality $y + 1 = x$ cannot be processed by merely placing $y + 1$ and x in the same equivalence class. This equality also implies that $y = x - 1$, $y - x = -1$, $x - y = 1$, $y + 3 = x + 2$, and so on. In order to avoid processing all these variations on the given equality, Shostak restricts his attention to *solvable* theories where an equality of the form $y + 1 = x$ can be solved for x to yield the solution $x = y + 1$. If the theories considered are also *canonizable*, then there is a canonizer σ such that whenever an equality $a = b$ is valid, then $\sigma(a) \equiv \sigma(b)$, where $\equiv$ represents syntactic equality. A canonizer for linear arithmetic can be defined to place terms into an ordered sum-of-monomials form. Once a solved form such as $x = y + 1$ has been obtained, all the other consequences $a = b$ of this equality can be obtained by $\sigma(a') = \sigma(b')$ where a' and b' are the results of substituting the solution for x into a and b, respectively. For example, substituting the solution into $y = x - 1$ yields $y = y + 1 - 1$, and the subsequent canonization step yields $y = y$.

The notion of a solvable and canonizable theory is extended to equalities involving a mix of interpreted and uninterpreted symbols by treating uninterpreted terms as variables. For the conjecture,

$$f(x-1)-1 = x+1,\ f(y)+1 = y-1,\ y+1 = x \vdash \mathit{false},$$

Shostak's algorithm would solve the equality $f(x-1)-1 = x+1$ as $f(x-1) = x+2$, the equality $f(y)+1 = y-1$ as $f(y) = y-2$, and $y+1 = x$ as $x = y+1$. Now, $f(x-1)$ and $f(y)$ are congruent because the canonical form for $x - 1$ obtained after substituting the solution $x = y + 1$ is y. By congruence closure, the equivalence classes of $f(x-1)$ and $f(y)$ have to be merged. In Shostak's original algorithm the current representatives of these equivalence classes, namely $x+2$ and $y - 2$ are merged. The resulting equality $x + 2 = y - 2$ is first solved to yield $x = y - 4$. This is incorrect because we already have a solution for x as $x = y+1$ and x should therefore have been eliminated. The new solution $x = y - 4$ contradicts the earlier one, but this contradiction goes undetected by Shostak's algorithm. This example can be easily adapted to show nontermination. Consider

$$f(v) = v, f(u) = u - 1, u = v \vdash \mathit{false}.$$

The merging of u and v here leads to the detection of the congruence between $f(u)$ and $f(v)$. This leads to solving of $u - 1 = v$ as $u = v + 1$. Now, the algorithm merges v and $v+1$. Since v occurs in $v+1$, this causes $v + 1$ to be merged with $v + 2$, and so on.

An earlier paper by Cyrluk, Lincoln, and Shankar [CLS96] gave an explanation (with minor corrections) of Shostak's algorithm for congruence closure and its extension to interpreted theories. Though proofs of correctness for the combination algorithm are briefly sketched, the algorithm presented there is both incomplete and nonterminating. Other published variants of Shostak's algorithm used in SVC [BDL96] and STeP [Bjø99] inherit these problems.

In this paper, we present an algorithm that fixes the incompleteness and nontermination in earlier versions of Shostak's algorithms. In the above example, the incompleteness is fixed by substituting the solution for x into the terms representing the different equivalence classes. Thus, when $f(x - 1)$ and $f(y)$ are detected to be congruent, their equivalence classes are represented by $y+3$ and $y-2$, respectively. The resulting equality $y+3 = y-2$ easily yields a contradiction. The nontermination is fixed by ensuring that no new mergeable

terms, such as $v+2$, are created during the processing of an axiom in T. Our algorithm is presented as a system of transformations on a set of equalities in order to capture the key insights underlying its correctness. We outline rigorous proofs for the termination, soundness, and completeness of this procedure. The algorithm as presented here emphasizes logical clarity over efficiency, but with suitable optimizations and data structures, it can serve as the basis for an efficient implementation. SRI's ICS (Integrated Canonizer/Solver) decision procedure package [FORS01] is directly based on the algorithm studied here.

Section 2 introduces the theory of equality, which is augmented in Section 3 with function symbols from a canonizable and solvable theory. Section 3 also introduces the basic building blocks for the decision procedure. The algorithm itself is described in Section 4 along with some example hand-simulations. The proofs of termination, soundness, and completeness are outlined in Section 5.

2 Background

With respect to a *signature* consisting of a set of function symbols F and a set of variables V, a term is either a variable x from V or an application $f(a_1,\ldots,a_n)$ of an n-ary function symbol f from F to n terms $a_1,\ldots,a_n$, where $0 \leq n$. The metavariable conventions are that u, v, x, y, and z range over variables, and a, b, c, d, and e range over terms. The metavariables R, S, and T, range over sets of equalities. The metatheoretic assertion $a \equiv b$ indicates that a and b are syntactically identical terms. Let $vars(a)$, $vars(a=b)$, and $vars(T)$ return the variables occurring in a term a, an equality $a=b$, and a set of equalities T, respectively. The operation $\lceil a \rceil$ is defined to return the set of all subterms of a.

Some of the function symbols are *interpreted*, i.e., they have a specific interpretation in some given theory τ, while the remaining function symbols are uninterpreted, i.e., can be assigned arbitrary interpretations. A term $f(a_1,\ldots,a_n)$ is interpreted (uninterpreted) if f is interpreted (uninterpreted). A term e is *non-interpreted* if it is either a variable or an uninterpreted term. We say that a term a *occurs interpreted* in a term e if there is an occurrence of a in e that is not properly within an uninterpreted subterm of e. Likewise, a *occurs uninterpreted* in e if a is a proper subterm of an uninterpreted subterm of e. $solvables(a)$ denotes the set of outermost non-interpreted subterms of a, i.e., those that do not occur uninterpreted in a.

$$solvables(f(a_1,\ldots,a_n)) = \bigcup_i solvables(a_i), \text{ if } f \text{ is interpreted}$$
$$solvables(a) = \{a\}, \text{ otherwise}$$

The theory of equality deals with sequents of the form $T \vdash a=b$. We will insist that these sequents be such that $vars(a=b) \subseteq vars(T)$. The proof theory for equality is given by the following inference rules.

1. Axiom: $\dfrac{}{T \vdash a=b}$, for $a=b \in T$.
2. Reflexivity: $\dfrac{}{T \vdash a=a}$.
3. Symmetry: $\dfrac{T \vdash a=b}{T \vdash b=a}$.
4. Transitivity: $\dfrac{T \vdash a=b \qquad T \vdash b=c}{T \vdash a=c}$.
5. Congruence: $\dfrac{T \vdash a_1=b_1 \quad \ldots \quad T \vdash a_n=b_n}{T \vdash f(a_1,\ldots,a_n)=f(b_1,\ldots,b_n)}$.

The semantics for terms is given by a model M over a domain D and an assignment ρ for the variables so that $M[\![x]\!]_\rho = \rho(x)$ and $M[\![f(a_1,\ldots,a_n)]\!]_\rho = M(f)(M[\![a_1]\!]_\rho,\ldots,M[\![a_n]\!]_\rho)$, and $M[\![a]\!]_\rho \in D$ for all a. We say that $M,\rho \models a=b$ iff $M[\![a]\!]_\rho = M[\![b]\!]_\rho$, and $M \models a=b$ iff $M,\rho \models a=b$ for all assignments ρ over $vars(a=b)$. We write $M,\rho \models S$ when $\forall a,b : a=b \in S \supset M,\rho \models a=b$, and $M,\rho \models T \vdash a=b$ when $(M,\rho \models T) \supset (M,\rho \models a=b)$.

3 Canonizable and Solvable Theories

Shostak's algorithm goes beyond congruence closure by deciding equality in the presence of function symbols that are *interpreted* in a theory τ [Sho84, CLS96]. The algorithm is targeted at canonizable and solvable theories, i.e., theories that are equipped with solvers and canonizers as outlined below. We write $\models_\tau a=b$ to indicate that $a=b$ is valid in theory τ. The canonizer and solver are first described for pure τ-terms, i.e., without any uninterpreted function symbols, and then extended to uninterpreted terms by regarding these as variables.

Definition 3.1 *A theory τ is* canonizable *if there is a canonizer σ such that*

1. $\models_\tau a = b$ *iff* $\sigma(a) \equiv \sigma(b)$.
2. $\sigma(x) \equiv x$.
3. $vars(\sigma(a)) \subseteq vars(a)$.
4. $\sigma(\sigma(a)) \equiv \sigma(a)$.
5. *If* $\sigma(a) \equiv f(b_1, \ldots, b_n)$, *then* $\sigma(b_i) \equiv b_i$ *for* $1 \leq i \leq n$.

For example, a canonizer σ for the theory of linear arithmetic can be defined to transform expressions into an ordered-sum-of-monomials normal form. A term a is said to be *canonical* if $\sigma(a) \equiv a$.

Definition 3.2 *A model M is a* σ-model *if $M \models a = \sigma(a)$ for any term a, and $M \not\models a = b$ for distinct canonical, variable-free terms a and b.*

Definition 3.3 *A set of equalities S and $a = b$ are* σ-equivalent *iff for all σ-models M and assignments ρ over the variables in a and b, $M, \rho \models a = b$ iff there is an assignment ρ' extending ρ, over the variables in S, a, and b, such that $M, \rho' \models S$.*

Definition 3.4 *A canonizable theory is* solvable *if there is an operation solve such that $solve(a = b) = \bot$ if $a = b$ is unsatisfiable in any σ-model, or $S = solve(a = b)$ for a set of equalities S such that*

1. *S is a set of n equalities of the form $x_i = e_i$ for $0 \leq n$ where for each i, $0 < i \leq n$,*
   (a) $x_i \in vars(a = b)$.
   (b) $x_i \notin vars(e_j)$, *for* j, $0 < j \leq n$.
   (c) $x_i \not\equiv x_j$, *for* $i \neq j$ *and* $0 < j \leq n$.
   (d) $\sigma(e_i) \equiv e_i$.
2. *S and $a = b$ are σ-equivalent.*

A solver for linear arithmetic, for example, takes an equation of the form

$$c + a_1x_1 + \ldots + a_nx_n = d + b_1x_1 + \ldots + b_nx_n,$$

where $a_1 \neq b_1$, and returns

$$\begin{aligned} x_1 = \sigma(\quad & (d - c)/(a_1 - b_1) \\ + \quad & ((b_2 - a_2)/(a_1 - b_1)) * x_2 \\ + \quad & \ldots \\ + \quad & ((b_n - a_n)/(a_1 - b_1)) * x_n). \end{aligned}$$

In general, $solve(a = b)$ may contain variables that do not occur in $a = b$, and vice-versa.

There are a number of interesting canonizable and solvable theories including linear arithmetic, the theory of tuples and projections, algebraic datatypes like lists, set algebra, and the theory of fixed-sized bitvectors. In many cases, the canonizability and solvability of the union of theories (with disjoint signatures) follows from the canonizability and solvability of its constituent theories.[1] We do not address modularity issues here but instead assume that we already have a canonizer and solver for a single combined theory.

The solvers and canonizers characterized above are intended to work in the absence of uninterpreted function symbols. They are adapted to terms containing uninterpreted subterms by treating these subterms as variables. Canonizers are applied to terms containing uninterpreted subterms by renaming distinct uninterpreted subterms with distinct new variables. For a given term a, let γ be a bijective mapping between a set of variables X that do not appear in a and the uninterpreted subterms of a. The application of a substitution γ to a term a, written $\gamma[a]$, is defined so that $\gamma[a] = f(\gamma[a_1], \ldots, \gamma[a_n])$ if $a \equiv f(a_1, \ldots, a_n)$, where f is interpreted. If a is in the domain of γ, then $\gamma[a] = \gamma(a)$, and otherwise, $\gamma[a] = a$. Then $\sigma(a)$ is $\gamma[\sigma(\gamma^{-1}[a])]$.

For solving equalities containing uninterpreted terms, we introduce, as with σ, a bijective map γ between a set of variables X not occurring in a or b, and the uninterpreted subterms of a and b, such that

$$solve(a = b) = \gamma[solve(\gamma^{-1}[a] = \gamma^{-1}[b])] \ .$$

When uninterpreted terms are handled as above, the conditions in Definitions 3.1 and 3.4 must be suitably adapted by using $solvables(a)$ instead of $vars(a)$.

The proof theory for equality is augmented for canonizable, solvable theories by the proof rules:

1. Canonization: $\dfrac{}{T \vdash a = \sigma(a)}$, for any term a.
2. Solve: $\dfrac{T \vdash a = b \quad T \cup S \vdash c = d}{T \vdash c = d}$ if $S = solve(a = b) \neq \bot$ and $vars(c = d) \subseteq vars(T)$.
3. Solve-$\bot$: $\dfrac{T \vdash a = b}{T \vdash false}$, if $solve(a = b) = \bot$.

A sequent $T \vdash c = d$ is derivable if there is a proof of $T \vdash c = d$ using one of the inference rules: axiom, reflexivity, symmetry, transitivity, congruence, canonization, solve, or solve-$\bot$. We say that $T \vdash S$ is derivable if $T \vdash c = d$ is derivable for every $c = d$ in S. The sequent $T, S \vdash c = d$ is just $T \cup S \vdash c = d$. The *weakening* and *cut* lemmas below are easily verified.

[1] The general result on combining solvers claimed by Shostak [Sho84] is incorrect, but there are some restricted results on combining equational unifiers [BS96] that can be applied here.

Lemma 3.5 (weakening) *If $T \subseteq T'$ and $T \vdash a = b$ is derivable, then $T' \vdash a = b$ is derivable.*

Lemma 3.6 (cut) *If $T' \vdash T$ and $T \vdash a = b$ is derivable, then $T' \vdash a = b$ is derivable.*

Theorem 3.7 (proof soundness) *If $T \vdash a = b$ is derivable, then for any σ-model M and assignment ρ over $vars(T)$, $M, \rho \models T \vdash a = b$.*

Proof. By induction on the derivation of $T \vdash a = b$. The soundness of the *solve* rules follows from the conditions in Definition 3.4. ∎

A set of equalities S is said to be *functional* (in a left-to-right reading of the equality) if whenever $a = b \in S$ and $a = b' \in S$, $b \equiv b'$. For example, the solution set returned by *solve* is functional. A functional set of equalities can be treated as a substitution and the associated operations are defined below. $S(a)$ returns the solution for a if it exists in S, and a itself, otherwise. If $a = b$ is in S for some b, then a is in the domain of S, i.e., $dom(S)$.

$$\begin{aligned} S(a) &= \begin{cases} b & \text{if } a = b \in S \\ a & \text{otherwise} \end{cases} \\ dom(S) &= \{a \mid \exists b.\ a = b \in S\}. \end{aligned}$$

The operation $a \overset{S}{\sim} b$ checks if a is congruent to b in S, i.e., $a \equiv f(a_1, \ldots, a_n)$, $b \equiv f(b_1, \ldots, b_n)$, and $S(a_i) \equiv S(b_i)$ for $1 \leq i \leq n$. A set of equalities S is said to be *congruence-closed* when for any terms a and b in $dom(S)$ such that $a \overset{S}{\sim} b$, we have $S(a) \equiv S(b)$.

$S[a]$ replaces a subterm b in a by $S(b)$, where $b \in solvables(a)$.

$$\begin{aligned} S[f(a_1, \ldots, a_n)] &= f(S[a_1], \ldots, S[a_n]), \\ &\quad \text{if } f \text{ is interpreted} \\ S[a] &= S(a), \text{ otherwise.} \end{aligned}$$

$norm(S)(a)$ is a normal form for a with respect to S and is defined as $\sigma(S[a])$. The operation *norm* does not appear in Shostak's algorithm and is the key element of our algorithm and its proof. With S fixed, we use $\hat{a}$ as a syntactic abbreviation for $norm(S)(a)$.

$$norm(S)(a) = \sigma(S[a]).$$

Lemma 3.8 *If $solve(a = b) = S \neq \bot$, then $norm(S)(a) \equiv norm(S)(b)$.*

Proof. By definitions 3.3 and 3.4(2), for any σ-model M and assignment ρ', we have $M, \rho' \models S \iff M, \rho' \models a = b$. Let $a' \equiv S[a]$ and $b' \equiv S[b]$. By induction on a, $M, \rho' \models a = a'$, and similarly $M, \rho' \models b = b'$. Hence, $M, \rho' \models a' = b'$. Then, since M is a σ-model, by Definition 3.2, it must be the case that $\sigma(a') \equiv \sigma(b')$, and therefore $norm(S)(a) \equiv norm(S)(b)$. ∎

The definition of the *lookup* operation uses Hilbert's epsilon operator, indicated by the keyword *when*, to return $S(f(b_1, \ldots, b_n))$ when $b_1, \ldots, b_n$ satisfying the listed conditions can be found. If no such $b_1, \ldots, b_n$ can be found, then $lookup(S)(a)$ returns a itself. We show later that the *lookup* operation is used only when the results of this choice are deterministic.

$$\begin{aligned} lookup(S)(f(a_1, \ldots, a_n)) &= S(f(b_1, \ldots, b_n)), \\ &\quad when\ b_1, \ldots, b_n : \\ &\quad f(b_1, \ldots, b_n) \in dom(S), \\ &\quad \text{and } a_i \equiv S(b_i), \\ &\quad \text{for } 1 \leq i \leq n \\ lookup(S)(a) &= a, \text{ otherwise.} \end{aligned}$$

$can(S)(a)$ is a canonical form in which any uninterpreted subterm e that is congruent to a known left-hand side e' in S is replaced by $S(e')$. It is analogous to the *canon* operation in Shostak's algorithm. We use $\overline{a}$ as a syntactic abbreviation for $can(S)(a)$.

$$\begin{aligned} can(S)(f(a_1, \ldots, a_n)) &= lookup(S)(f(\overline{a_1}, \ldots, \overline{a_n})), \\ &\quad \text{if } f \text{ is uninterpreted} \\ can(S)(f(a_1, \ldots, a_n)) &= \sigma(f(\overline{a_1}, \ldots, \overline{a_n})), \\ &\quad \text{if } f \text{ is interpreted} \\ can(S)(a) &= S(a), \text{ otherwise.} \end{aligned}$$

Lemma 3.9 (σ-norm) *If S is functional, then $norm(S)(\sigma(a)) \equiv \hat{a}$ and $can(S)(\sigma(a)) \equiv \overline{a}$.*

Proof. We know that $\vdash \sigma(a) = a$. Then for $b' = S[\sigma(a)]$ and $b = S[a]$, the equality $b' = b$ is valid in every σ-model. Then by Definition 3.2, $\sigma(S[\sigma(a)]) \equiv \sigma(S[a])$, and hence the first part of the theorem.

The reasoning in the second part is similar. If we let $R = \{b = \overline{b} \mid b \in [\![a]\!]\}$, then $can(S)(a) \equiv norm(R)(a)$. We can therefore use the first part of the theorem to establish the second part. ∎

We next introduce a composition operation for merging the results of a *solve* operation into an existing solution set. When $R \circ S$ is used, S must be functional, and the result contains $a = \hat{b}$ for each equality $a = b$ in R in addition to the equalities in S.

$$R \circ S = \{a = \hat{b} \mid a = b \in R\} \cup S.$$

The following lemmas about composition are given without proof.

Lemma 3.10 (norm decomposition) *If $R \cup S$ is functional, then*

$$norm(R \circ S)(a) \equiv norm(S)(norm(R)(a)).$$

$$
\begin{aligned}
process(\{a = b, T\}) &= assert(a = b, process(T)) \\
process(\emptyset) &= \emptyset. \\
assert(a = b, \bot) &= \bot \\
assert(a = b, S) &= cc(merge(\overline{a}, \overline{b}, S^+)), \text{ where,} \\
&\quad S^+ = expand(S, \overline{a}, \overline{b}). \\
expand(S, a, b) &= S \cup \{e = e \mid e \in new(S, a, b)\}. \\
new(S, a, b) &= [\![a = b]\!] - dom(S). \\
merge(a, b, S) &= \bot, \text{if } solve(a = b) = \bot \\
merge(a, b, S) &= S \circ solve(a = b), \text{ otherwise.} \\
cc(\bot) &= \bot \\
cc(S) &= cc(merge(S(a), S(b), S)), \\
&\quad when\ a, b : \\
&\quad a, b \in dom(S) \\
&\quad a \overset{S}{\sim} b, \text{ and } S(a) \not\equiv S(b) \\
cc(S) &= S, \text{ otherwise.}
\end{aligned}
$$

Figure 1: Main Procedure: *process*

Lemma 3.11 (associativity of composition) *If $Q \cup R \cup S$ is functional, then*

$$(Q \circ R) \circ S = Q \circ (R \circ S).$$

Lemma 3.12 (monotonicity) *If $R \cup S$ is functional, then if $R(a) \equiv R(b)$, then $(R \circ S)(a) \equiv (R \circ S)(b)$, for any a and b.*

4 An Algorithm for Deciding Equality in the Presence of Theories

We next present an algorithm for deciding $T \vdash c = d$ for terms containing uninterpreted function symbols and function symbols interpreted in a canonizable and solvable theory. The algorithm for verifying $T \vdash c = d$ checks that $can(S)(c) \equiv can(S)(d)$, where $S = process(T)$. The *process* procedure shown in Figure 1, is written as a functional program. It is a mathematical description of the algorithm and not an optimized implementation. The *state* of the algorithm consists of a set of equalities S which holds the solution set. We demonstrate as an invariant that S is functional. Two terms a and b in $dom(S)$ are in the same equivalence class according to S if $S(a) \equiv S(b)$.

The operation $process(T)$ returns a final solution set by starting with an empty solution set and successively processing each equality $a = b$ in T by invoking $assert(a = b, S)$, where S is the state as returned by the recursive call of *process*. The invocation of $assert(a = b. S)$ is executed by first reducing a and b to their respective canonical forms $\overline{a}$ and $\overline{b}$. Next, S is expanded to include $e = e$ for each subterm e of $\overline{a} = \overline{b}$ where $e \notin dom(S)$. This preprocessing step ensures that S contains entries corresponding to any terms that might be needed in the congruence closure phase in the operation *cc*.[2] The *merge* operation then solves the equality $\overline{a} = \overline{b}$ to get a solution[3] S', and returns $S \circ S'$ as the new value for the state S. As we will show, this new value affirms $a = b$, but it is not congruence-closed and hence does not contain all the consequences of the assertion $a = b$. The step $cc(S)$ computes the congruence closure of S by repeatedly picking a pair of congruent terms a and b from $dom(S)$ such that $S(a) \not\equiv S(b)$ and merging them using $merge(S(a), S(b), S)$. Eventually either a contradiction is found or all congruent left-hand sides in S are merged and the *cc* operation terminates returning a congruence-closed solution set.

The above algorithm fixes the nontermination and incompleteness problems in Shostak's algorithm by introducing the *norm* operation and the composition operator $R \circ S$ to fold in a solution. The *norm* operation ensures that no new uninterpreted terms are introduced during congruence closure in the function *cc*, as is needed to guarantee termination. The composition operator $R \circ S$ ensures that any newly generated solution S is immediately substituted into R and the algorithm never attempts to find a solution for an already solved non-interpreted term.

We first illustrate the algorithm on some examples. The first example contains no interpreted symbols.

Example 4.1 Consider the goal $f^5(x) = x, f^3(x) = x \vdash f(x) = x$. The value of S after the base case is $\emptyset$. After the preprocessing of $f^3(x) = x$ in *assert*, S is $\{x = x, f(x) = f(x), f^2(x) = f^2(x), f^3(x) = f^3(x)\}$. After merging $f^3(x)$ and x, S is $\{x = x, f(x) = f(x), f^2(x) = f^2(x), f^3(x) = x\}$. When $f^5(x) = x$ is preprocessed in *assert*, $can(S)(f^5(x))$ yields $f^2(x)$ since $S(f^3(x)) \equiv x$, and S is left unchanged. When $f^2(x)$ and x have been merged, S is $\{x = x, f(x) = f(x), f^2(x) = x, f^3(x) = x\}$. Now $f(x) \overset{S}{\sim} f^3(x)$ and hence $f(x)$ and x are merged so that S is now $\{x = x, f(x) = x, f^2(x) = x, f^3(x) = x\}$.

[2] Actually, the interpreted subterms of $\overline{a} = \overline{b}$ need not all be included in $dom(S)$. Only those that are immediate subterms of uninterpreted subterms in $\overline{a} = \overline{b}$ are needed.

[3] Any variables occurring in $solve(a = b)$ and not in $vars(a = b)$ must be fresh, i.e., they must not occur in the original conjecture or be generated by any other invocation of *solve*.

The conclusion $f(x) = x$ easily follows since $can(S)(f(x)) \equiv x \equiv can(S)(x)$.

Example 4.2 Consider $y + 1 = x$, $f(y) + 1 = y - 1$, $f(x-1) - 1 = x + 1 \vdash$ *false* which is a permutation of our earlier example. Starting with $S \equiv \emptyset$ in the base case, the preprocessing of $f(x-1) - 1 = x + 1$ causes the equation to be placed into canonical form as $-1 + f(-1 + x) = 1 + x$ and S is set to

$$\{\ 1 = 1, -1 = -1, x = x, -1 + x = -1 + x,\\ f(-1 + x) = f(-1 + x), 1 + x = 1 + x\}.$$

Solving $-1+f(-1+x) = 1+x$ yields $f(-1+x) = 2+x$, and S is set to

$$\{\ 1 = 1, -1 = -1, x = x, -1 + x = -1 + x,\\ f(-1 + x) = 2 + x, 1 + x = 1 + x\}.$$

No unmerged congruences are detected. Next, $f(y) + 1 = y - 1$ is asserted. Its canonical form is $1 + f(y) = -1 + y$, and once this equality is asserted, the value of S is

$$\{\ 1 = 1, -1 = -1, x = x, -1 + x = -1 + x,\\ f(-1 + x) = 2 + x, 1 + x = 1 + x, y = y,\\ f(y) = -2 + y, -1 + y = -1 + y,\\ 1 + f(y) = -1 + y\}.$$

Next $y + 1 = x$ is processed. Its canonical form is $1 + y = x$ and the equality $1 + y = 1 + y$ is added to S. Solving $y + 1 = x$ yields $x = 1 + y$, and S is reset to

$$\{\ 1 = 1, -1 = -1, x = 1 + y, -1 + x = y,\\ f(-1 + x) = 3 + y, 1 + x = 2 + y, y = y,\\ f(y) = -2 + y, -1 + y = -1 + y,\\ 1 + f(y) = -1 + y, 1 + y = 1 + y\}.$$

The congruence close operation *cc* detects the congruence $f(1 - y) \overset{S}{\sim} f(x)$ and invokes *merge* on $3 + y$ and $-2 + y$. Solving this equality $3 + y = -2 + y$ yields $\perp$ returning the desired contradiction.

5 Analysis

We describe the proofs of termination, soundness, and completeness, and also present a complexity analysis.

Key Invariants. The *merge* operation is clearly the workhorse of the procedure since it is invoked from within both *assert* and *cc*. Let $U(X)$ represent the set $\{a \in X \mid a \text{ uninterpreted}\}$ of uninterpreted terms in the set X. Let A be $solvables(a)$, B be $solvables(b)$, and $S' = merge(a, b, S)$, then assuming $U(A \cup B) \subseteq dom(S)$ and for all $c \in A \cup B$, $S(c) \equiv c$, the following properties hold of S' if they hold of S:

1. Functionality.
2. Subterm closure: S is *subterm-closed* if for any $a \in dom(S)$, $[\![a]\!] \subseteq dom(S)$.
3. Range closure: S is *range-closed* if for any $a \in dom(S)$, $U(solvables(S(a))) \subseteq dom(S)$, and for any $c \in solvables(S(a))$, $S(c) \equiv c$.
4. Norm closure: S is *norm-closed* if $S(a) \equiv norm(S)(a)$ for a in $dom(S)$. This of course holds trivially for uninterpreted terms a.
5. Idempotence: S is *idempotent* if $S[S(a)] \equiv S(a)$, $norm(S)(S(a)) \equiv S(a)$, and $norm(S)(norm(S)(a)) \equiv norm(S)(a)$.

These properties can be easily established by inspection. Since whenever $merge(a, b, S)$ is invoked in the algorithm, the arguments do satisfy the conditions $U(A \cup B) \subseteq dom(S)$ and for all $c \in A \cup B$, $S(c) \equiv c$, it then follows that these properties are also preserved by *assert* and *cc*, and therefore hold of $process(T)$. We assume below that these invariants hold of S whenever the metavariable S is used with or without subscripts or superscripts.

Lemma 5.1 (merge equivalence) *Let $A = solvables(a)$ and $B \equiv solvables(b)$. Given that $U(A \cup B) \subseteq dom(S)$ and for all $c \in A \cup B$, $S(c) \equiv c$, if $S' = merge(a, b, S) \neq \perp$, then*

1. $norm(S')(a) \equiv norm(S')(b)$.
2. $U(dom(S')) = U(dom(S))$.

Proof. Let $R \equiv solve(a = b)$. By definition, $merge(a, b, S) \equiv S \circ R$. By Lemma 3.8, $norm(R)(a) \equiv norm(R)(b)$. Since $S(c) \equiv c$ for $c \in A \cup B$, $norm(S)(a) \equiv a$ and $norm(S)(b) \equiv b$. Hence, by *norm decomposition*, we have $norm(S')(a) \equiv norm(S')(b)$.

By Definition 3.4, $dom(R) \subseteq A \cup B$, hence $U(dom(S')) = U(dom(S))$. ∎

Termination. We define $\#(S)$ to represent the number of distinct equivalence classes partitioning $U(dom(S))$ as given by $P(S)$.

$$\begin{aligned} E(S)(a) &= \{b \in U(dom(S)) \mid S(b) \equiv S(a)\} \\ P(S) &= \{E(S)(a) \mid a \in U(dom(S))\} \\ \#(S) &= |P(S)| \end{aligned}$$

The definition of $cc(S)$ terminates because the measure $\#(S)$ decreases with each recursive call. If in the definition of *cc*, $merge(S(a), S(b), S) = \bot$, then clearly *cc* terminates. Otherwise, let $S' = merge(S(a), S(b), S) \neq \bot$, for a and b in $dom(S)$ such that $S(a) \not\equiv S(b)$ and $a \overset{S}{\sim} b$. In this case a and b must be uninterpreted terms since for interpreted terms a and b, if $a \overset{S}{\sim} b$, then $S(a) \equiv S(b)$ by *norm closure*. By *merge equivalence*, $norm(S')(S(a)) \equiv norm(S')(S(b))$ and $U(dom(S')) = U(dom(S))$. By *monotonicity*, for any c and d such that $S(c) \equiv S(d)$, we have $S'(c) \equiv S(d)$, and therefore $\#(S') \leq \#(S)$. However, by *norm closure*, $S'(a) \equiv S'(b)$ so that $\#(S') < \#(S)$.

Soundness. The following lemmas establish the soundness of the operations *norm* and *can* with respect to S. *Substitution soundness* and *can soundness* are proved by a straightforward induction on a, and *norm soundness* is a simple consequence of *substitution soundness*.

Lemma 5.2 (substitution soundness)
If $vars(a) \subseteq vars(T \cup S)$, *then* $T, S \vdash a = a'$ *is derivable, for* $a' \equiv S[a]$.

Lemma 5.3 (norm soundness)
If $vars(a) \subseteq vars(T \cup S)$, *then* $T, S \vdash a = \hat{a}$ *is derivable.*

Lemma 5.4 (can soundness)
If $vars(a) \subseteq vars(T \cup S)$, *then* $T, S \vdash a = \tilde{a}$ *is derivable.*

Lemma 5.5 (merge soundness)
If $S' = merge(a, b, S) \neq \bot$, *then if* $T, S \vdash a = b$, *and* $T, S' \vdash c = d$ *with* $vars(c = d) \subseteq vars(T \cup S)$, *then* $T, S \vdash c = d$. *Otherwise,* $merge(a, b, S) = \bot$, *and* $T, S \vdash \bot$.

Proof. If $S' = merge(a, b, S) \neq \bot$, then let $R = solve(a = b)$. By *norm soundness*, $S, R \vdash S'$, and hence by *cut*, $T, S, R \vdash c = d$ is derivable. By the *solve* rule, $T, S \vdash c = d$ is derivable.

If $merge(a, b, S) = \bot$, then by similar reasoning using the *solve*-$\bot$ rule, $T, S \vdash$ *false* is derivable. ∎

Lemma 5.6 (cc soundness) *If* $S' = cc(S) \neq \bot$, $T, S' \vdash a = b$ *for* $vars(a = b) \subseteq vars(T, S)$, *then* $T, S \vdash a = b$ *is derivable. Otherwise,* $cc(S) = \bot$, *and* $S \vdash false$ *is derivable.*

Proof. By computation induction on the definition of *cc* using *merge soundness*. ∎

Lemma 5.7 (process soundness)
If $S = process(T_1) \neq \bot$, $T_1 \subseteq T_2$, *and* $T_2, S \vdash c = d$ *for* $vars(c = d) \subseteq vars(T_2)$, *then* $T_2 \vdash c = d$ *is derivable. Otherwise,* $process(T_1) = \bot$, *and* $T_1 \vdash$ *false is derivable.*

Proof. By induction on the length of T_1. In the base case, S is empty and the theorem follows trivially. In the induction step, with $T_1 = \{a = b, T_1'\}$ and $S' = process(T_1')$, we have the induction hypothesis that $T_2 \vdash c = d$ is derivable if $T_2, S' \vdash c = d$ is derivable, for any c, d such that $vars(c = d) \subseteq vars(T_2)$. We know by *can soundness* that $T_2, S' \vdash \overline{a} = a$ and $T_2, S' \vdash \overline{b} = b$ are derivable. When S' is augmented with identities over subterms of $\overline{a}$ and $\overline{b}$ to get S'^{+}, we have the derivability of $T_2, S' \vdash S'^{+}$. By *cc soundness*, we then have the derivability of $T_2, S'^{+} \vdash c = d$ from that of $T_2, S \vdash c = d$. The derivability of $T_2, S' \vdash c = d$ then follows by *cut* from that of $T_2, S'^{+} \vdash c = d$, and we get the conclusion $T_2 \vdash c = d$ by the induction hypothesis.

A similar induction argument shows that when $process(T_1) = \bot$, then $T_2 \vdash$ *false*. ∎

Theorem 5.8 (soundness) *If* $S = process(T) \neq \bot$, $vars(a = b) \subseteq vars(T)$, *and* $\overline{a} \equiv \overline{b}$, *then* $T \vdash a = b$ *is derivable. Otherwise,* $process(T) = \bot$, *and* $T \vdash$ *false is derivable.*

Proof. If $S = process(T) \neq \bot$, then by *can soundness*, $T, S \vdash a = \overline{a}$ and $T, S \vdash b = \overline{b}$ are derivable. Hence, by transitivity and symmetry, $T, S \vdash a = b$ is derivable. Therefore, by *process soundness*, $T \vdash a = b$ is derivable.

If $process(T) = \bot$, then already by *process soundness*, $T \vdash$ *false*. ∎

Completeness. We show that when $S = process(T)$ then $can(S)$ is a σ-model satisfying T. When this is the case, completeness follows from *proof soundness*. In proving completeness, we exploit the property that the output of *process* is congruence-closed.

Lemma 5.9 (confluence)
If S *is congruence-closed and* $U(\llbracket a \rrbracket) \subseteq dom(S)$, *then* $can(S)(a) \equiv norm(S)(a)$.

Proof. The proof is by induction on a. In the base case, when a is a variable, $can(S)(a) \equiv S(a) \equiv norm(S)(a)$.

If a is uninterpreted and of the form $f(a_1, \ldots, a_n)$, then $can(S)(a) \equiv lookup(S)(f(\overline{a_1}, \ldots, \overline{a_n}))$. Since S is *subterm-closed*, by the induction hypothesis and *norm closure*, we have $\overline{a_i} \equiv \hat{a}_i \equiv S(a_i)$ for $0 < i \leq n$. Then

there must be some b of the form $f(b_1, \ldots, b_n)$ such that $S(b_i) \equiv S(a_i)$, for $0 < i \leq n$, since a itself is such a b. Then by *congruence closure* and *norm closure*, $\overline{a} \equiv S(b) \equiv S(a) \equiv \hat{a}$, since $a \overset{S}{\sim} b$.

If a is interpreted, by the induction hypothesis and *subterm closure*, $\overline{a} \equiv \sigma(f(\overline{a_1}, \ldots, \overline{a_n})) \equiv \sigma(f(\hat{a_1}, \ldots, \hat{a_n})) \equiv \hat{a}$. ∎

Lemma 5.10 (can composition) *If $S' = S \circ R$ and S' is congruence-closed, then $can(S')(can(S)(a)) \equiv can(S')(a)$.*

Proof. By induction on a. When a is a variable. $can(S)(a) \equiv S(a)$. If $a \notin dom(S)$, then $S(a) = a$, and hence the conclusion. Otherwise, by range-closure, $U(\llbracket S(a) \rrbracket) \subseteq dom(S) \subseteq dom(S')$. Then, by *confluence*, *norm decomposition*, and *idempotence*, $can(S')(S(a)) \equiv norm(S')(S(a)) \equiv norm(R)(norm(S)(S(a))) \equiv norm(R)(norm(S)(a)) \equiv norm(S')(a) \equiv can(S')(a)$.

In the induction step, let $a \equiv f(a_1, \ldots, a_n)$. If a is uninterpreted, then if

$$f(\overline{a_1}, \ldots, \overline{a_n}) \overset{S}{\sim} f(b_1, \ldots, b_n)$$

for some $f(b_1, \ldots, b_n) \in dom(S)$, then $\overline{a} \equiv S(f(b_1, \ldots, b_n))$. The reasoning used in the base case can then be repeated to derive the conclusion. Otherwise, $\overline{a} \equiv f(\overline{a_1}, \ldots, \overline{a_n})$ and by the induction hypothesis and the definition of *can*, $can(S')(\overline{a}) \equiv lookup(S')(f(can(S')(a_1), \ldots, can(S')(a_n))) \equiv can(S')(a)$.

When a is interpreted, by the induction hypothesis and the *σ-norm* lemma,

$$\begin{aligned} & can(S')(\overline{a}) \\ \equiv\ & can(S')(\sigma(f(\overline{a_1}, \ldots, \overline{a_n}))) \\ \equiv\ & \sigma(f(can(S')(\overline{a_1}), \ldots, can(S')(\overline{a_n}))) \\ \equiv\ & can(S')(a). \end{aligned}$$

∎

Lemma *can composition* with $\emptyset$ for R yields the idempotence of $can(S)$ for congruence-closed S so that we can define a σ-model M_S in terms of $can(S)$. The domain D of M_S consists of $\{a | can(S)(a) = a\}$. The mapping of functions is such that $M_S(f)(\mathbf{a_1}, \ldots, \mathbf{a_n}) = lookup(S)(f(\mathbf{a_1}, \ldots, \mathbf{a_n}))$, if f is uninterpreted. If f is interpreted $M_S(f)(\mathbf{a_1}, \ldots, \mathbf{a_n}) = \sigma(f(\mathbf{a_1}, \ldots, \mathbf{a_n}))$. If $\rho[x] = \rho(x)$ and $\rho[f(a_1, \ldots, a_n)] = f(\rho[a_1], \ldots, \rho[a_n])$, then by the idempotence of $can(S)$, $M_S\llbracket a \rrbracket_\rho$ is just $can(S)(\rho[a])$. Lemma σ-norm can then be used to show $M_S \models \sigma(a) = a$. M_S is therefore a σ-model. Correspondingly, for a given set of variables X, ρ_S^X is defined so that $\rho_S^X(x) = can(S)(x)$ for $x \in X$.

Lemma 5.11 (can σ-model) *If $S = process(T) \neq \bot$ and $X = vars(T)$, then $M_S, \rho_S^X \models a = b$ for any $a = b \in T$.*

Proof. Showing that $M_S, \rho_S^X \models a = b$ is the same as showing that $can(S)(a) \equiv can(S)(b)$. The proof is by induction on T. In the base case, T is empty. In the induction step, $T = \{a = b, T'\}$ with $X' = vars(T')$. Let $S' = process(T')$. By the induction hypothesis, $M_{S'}, \rho_{S'}^{X'} \models T'$. With $S'^+ = expand(S, a', b')$ for $a' \equiv can(S')(a)$ and $b' \equiv can(S')(b)$, let $S_0 = merge(a, b, S'^+)$, hence by *merge equivalence*, $norm(S_0)(a') \equiv norm(S_0)(b')$. By associativity of composition, it can be shown that there is an R such that $S = S_0 \circ R$ and an R' such that $S = S'^+ \circ R'$. Hence by monotonicity, $norm(S)(a') \equiv norm(S)(b')$. Since S is congruence-closed, by *confluence*, $can(S)(a') \equiv norm(S)(a')$ and $can(S)(b') \equiv norm(S)(b')$. Hence, $can(S)(a') \equiv can(S)(b')$.

It can also be shown that $can(S'^+)(a) \equiv can(S')(a)$, and similarly for b. Therefore, by *can composition*, we have $can(S)(a) \equiv can(S)(b)$, and hence $M_S, \rho_S^X \models a = b$. A similar argument shows that for $c = d \in T'$, since $can(S')(c) \equiv can(S')(d)$, we also have $can(S)(c) \equiv can(S)(d)$. ∎

When $T \vdash$ *false* is derivable, we know by *proof soundness* that there is no σ-model satisfying T and hence by the *can σ-model* lemma, $process(T)$ must be $\bot$.

Theorem 5.12 (completeness) *If $S = process(T) \neq \bot$ and $T \vdash a = b$, then $can(S)(a) \equiv can(S)(b)$.*

Proof. Since $M_S, \rho_S^X \models T$ by *can σ-model* for $X = vars(T)$, we have by *proof soundness* that $can(S)(a) \equiv can(S)(b)$. ∎

Complexity. We have already seen in the termination argument that the number of iterations of *cc* in *process* is bounded by the number of distinct equivalence classes of terms in $dom(S)$ which is no more than the number of distinct uninterpreted terms. We will assume that the *solve* operation is performed by an oracle and that there is some fixed bound on the size of the solution set returned by it. In the case of linear arithmetic, the solution set has at most one element. Let n represent the number of distinct terms appearing in T which is also a bound on $|S|$ and on the size of the largest term appearing in S. The composition operation can be implemented in linear time. Thus the entire algorithm has $O(n^2)$ steps assuming that the σ and *solve* operations are length-preserving and ignoring the time spent inside *solve*.

6 Conclusions

Shostak's decision procedure for equality in the presence of interpreted and uninterpreted functions is seriously flawed. It is both incomplete and nonterminating, and hence not a decision procedure. All subsequent variants of Shostak's algorithm have been similarly flawed. This is unfortunate because decision procedures based on Shostak's algorithm are at the core of a number of widely used verification systems. We have presented a correct algorithm that captures Shostak's key insights, and described proofs of termination, soundness, and completeness.

Acknowledgments: We are especially grateful to Clark Barrett for instigating this work and correcting several significant errors in earlier drafts, and to Jean-Christophe Filliâtre for his oCaml implementation which yielded useful feedback on the algorithm studied here. The presentation has been substantially improved thanks to the suggestions of the anonymous referees and those of Nikolaj Bjørner, David Cyrluk, Bruno Dutertre, Ravi Hosabettu, Pat Lincoln, Ursula Martin, David McAllester, Sam Owre, John Rushby, and Ashish Tiwari.

References

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.

[Bjø99] Nikolaj Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1999.

[BRRT99] L. Bachmair, C. R. Ramakrishnan, I.V. Ramakrishnan, and A. Tiwari. Normalization via rewrite closures. In *International Conference on Rewriting Techniques and Applications, RTA '99*, Berlin, 1999. Springer-Verlag.

[BS96] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. Symbolic Computation*, 21:211–243, 1996.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[EHD93] Computer Science Laboratory, SRI International, Menlo Park, CA. *User Guide for the* EHDM *Specification Language and Verification System, Version 6.1*, February 1993. Three volumes.

[FORS01] J-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In *CAV 01: Computer-Aided Verification*. Springer-Verlag, 2001. To appear.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *International Conference on Rewriting Techniques and Applications, RTA '97*, number 1232 in Lecture Notes in Computer Science, pages 23–37, Berlin, 1997. Springer-Verlag.

[MT96] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583 585, July 1978.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SSMS82] R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.

A Decision Procedure for an Extensional Theory of Arrays

Aaron Stump, Clark W. Barrett, and David L. Dill
Computer Systems Laboratory
Stanford University, Stanford, CA 94305, USA
E-mail: {stump,dill,barrett}@cs.stanford.edu

Jeremy Levitt
0-In Design Automation, Inc.
San Jose, CA 95110, USA
Email: levitt@0-In.com

Abstract

A decision procedure for a theory of arrays is of interest for applications in formal verification, program analysis, and automated theorem-proving. This paper presents a decision procedure for an extensional theory of arrays and proves it correct.

1. Introduction

A decision procedure for a theory of arrays is of interest for applications in formal verification and program analysis. Such a procedure is also of value for theorem-provers. The PVS theorem-prover [11] has an undocumented decision procedure for a theory of arrays [12], and HOL has some automatic support for a theory of arrays via a library for finite partial functions [3].

Two kinds of array theories have been studied previously. Extensional theories require that if two arrays store the same value at index i, for each index i, then the arrays must be the same. Non-extensional theories do not make this requirement. This paper is the first to present a procedure for checking satisfiability of arbitrary quantifier-free formulas in an extensional theory of arrays and prove its correctness.

2. Theories of arrays

Decision procedures for various theories of arrays have been studied previously. Most of these theories can be divided into extensional and non-extensional varieties. In this section, several families of array theories are axiomatized in classical first-order multi-sorted logic with equality. The theory **Arr** decided in this paper is then presented and compared to previously decided theories.

2.1. The language

Sorts The language has a basic sort I for indices into arrays. It also has value sorts, which are the sorts of individuals that may be stored in arrays. The sort V is the sort for primitive values stored in arrays. The set of value sorts is defined to be the least set X satisfying

- $V \in X$
- $\tau \in X \rightarrow array_\tau \in X$

Every value sort except V is an array sort. The value sorts together with I are all the sorts of the language. V and I need not be distinct.

Definition 1 (dimensionality of a value sort) *The dimension $dim(\tau)$ of a value sort τ is defined by*

- $dim(V) = 0$
- $dim(array_\tau) = dim(\tau) + 1$

Terms The language has countably infinitely many variables and constants, with countably infinitely many of each distinct sort. The constants are uninterpreted, in the sense they will not occur in any axiom or axiom scheme. The function symbols of the language are

- $read_\tau$ of type $(array_\tau \rightarrow I \rightarrow \tau)$, for every value sort τ
- $write_\tau$ of type $(array_\tau \rightarrow I \rightarrow \tau \rightarrow array_\tau)$, for every value sort τ

Subscripts on *read* and *write* will generally be omitted. Informally, $read(a, i)$ will denote the value stored in array a at index i, and $write(a, i, v)$ will denote an array which stores the same value as a for every index except possibly i, where it stores value v.

Terms are built up in the usual way from constants and variables using the function symbols. Terms whose sort is an array sort will be called array terms. Terms whose sort is I will be called index terms. The dimension $dim(a)$ of an array term a is the dimension of its sort. If $dim(a) = n$, array a is said to be n-dimensional. If $n > 1$, a is also said to be multi-dimensional.

0-7695-1281-X/01 $10.00 © 2001 IEEE

Formulas The atomic formulas of the language are the equations between terms of the same sort. Formulas are built up from atomic formulas using propositional connectives and quantifiers in the usual way. A formula is *closed* if it has no free variables. A *literal* is an atomic formula or the negation of an atomic formula. A *theory* is a set of closed formulas.

2.2. Theories

Some theories restrict which array sorts are allowed. If a theory allows array sorts of dimension at most n, it is said to have just n-dimensional arrays. If a theory allows all array sorts, it is said to have multi-dimensional arrays.

The following scheme, which is schematic in a value sort τ, is called the read-over-write axiom scheme. Informally, it says that for all arrays a, indices i and j, and values v of suitable type, reading the value stored at index j of $write(a, i, v)$ is v if the two indices are equal and $read(a, j)$ if they are different.

Axiom scheme 1 (read-over-write)

$$\forall a : array_\tau \,.\, \forall i : I \,.\, \forall j : I \,.\, \forall v : V \,.$$
$$(i = j \rightarrow read(write(a, i, v), j) = v) \wedge$$
$$(i \neq j \rightarrow read(write(a, i, v), j) = read(a, j))$$

The following scheme, which is schematic in a value sort τ, is called the extensionality axiom scheme. Informally, it expresses a principle of extensionality for arrays: if two arrays store the same value at index i, for each index i, they are equal.

Axiom scheme 2 (extensionality)

$$\forall a : array_\tau \,.\, \forall b : array_\tau \,.$$
$$(\forall i : I \,.\, read(a, i) = read(b, i)) \rightarrow a = b$$

The extensional theories are those axiomatized by the read-over-write and extensionality axiom schemes. The non-extensional theories are those axiomatized by just the read-over-write axiom scheme. Note that since a theory is a set of closed formulas, quantifier-free array theories have no variables; all 0-ary symbols are (uninterpreted) constants.

2.3. The theory Arr

The theory **Arr** decided in this paper is the quantifier-free fragment of the extensional theory with multi-dimensional arrays where sort V is defined to be sort I. So indices are the values stored in 1-dimensional arrays.

The restriction to the quantifier-free fragment is justified by the fact that the fully quantified theory is undecidable, even in the absence of the function symbols $write_\tau$ and the read-over-write scheme. This is because single-sorted first-order theories with function symbols and equality may be translated into this array theory in such a way that a first-order formula is valid iff its translation is. The translation maps constant symbols to index constants, n-ary function symbols to n-dimensional array constants, and terms like $f(i_1, \ldots, i_n)$ to nested read expressions $read(\ldots read(read(f', i'_1), i'_2) \ldots, i'_n)$, where $f', i'_1, \ldots, i'_n$ are the translations of $f, i_1, \ldots, i_n$. The undecidability results for classical first order logic with just function symbols and equality (see, e.g., [5]) can then be applied to show that even quite restricted quantified fragments of the extensional theory of arrays are undecidable.

A decision procedure for **Arr** may be useful even for applications which require a fully quantified logic. Many theorem provers, such as the widely used PVS [11], provide strategies to reduce goals to subgoals in decidable fragments of their logic.

2.4. Comparison with related work

In this section, related work is summarized by describing which theories are decided. These theories often use axiomatizations different from but equivalent to that of **Arr**. All the theories decided are quantifier-free. Kaplan is the only one to distinguish the sorts V and I. Many of the previous theories allow arithmetic operators or uninterpreted functions over sort I to be used in addition to the symbols *read* and *write*. The restriction here to just the essential theory of arrays is justified by the fact that, as will be shown in Section 6 below, the satisfiability procedure for **Arr** is suitable for incorporation into a framework for cooperating decision procedures [2]. In such a framework, separate decision procedures for arithmetic and uninterpreted functions may be combined with the decision procedure for **Arr** to decide the combined theory.

The first two works present axioms but no decision procedure for their theories. With the exception of Levitt's work, the others give decision procedures for theories that are strictly weaker than **Arr**, either because they restrict the form of formulas in the theory (e.g., to just equations), disallow equations between arrays, or are non-extensional.

McCarthy In [8], McCarthy introduces the function symbols *read* and *write* and gives an informal semantics for an extensional theory of arrays based on them.

Collins and Syme Collins and Syme present in HOL a theory of finite higher-order partial functions similar to a theory with multi-dimensional arrays [3].

Kaplan In [6], Kaplan gives a decision procedure for a non-extensional equational theory with just 1-dimensional arrays. He considers equations between index terms only, which is reasonable since his theory contains no non-trivial equations between arrays. He then shows how to extend his

procedure to decide an extensional equational theory, where the equations may be between array as well as index terms. He imposes the restriction that distinct variables of sort I must receive distinct interpretations.

Suzuki and Jefferson In [15], Suzuki and Jefferson present a decision procedure for a theory with just 1-dimensional arrays, where equations between arrays are not allowed. The theory has axioms for extensionality and the existence of constant arrays (arrays that store the same value at all indices), but these appear to be included for technical reasons only; the theory decided is equivalent to the one without those axioms under the restrictions they impose. They extend their procedure to decide a theory with a new predicate symbol *PERM*, where $PERM(a, b)$ holds iff the multiset of the values stored in a is contained in the multiset of the values stored in b. Sentences of the theory are restricted to the form $P \rightarrow PERM(a, b)$, where P is any (quantifier-free) sentence not containing *PERM*. **Arr** does not have the *PERM* predicate, but inspection of the way Suzuki and Jefferson extend their algorithm to treat *PERM* shows that it could just as easily be used to extend the algorithm for **Arr**, as long as their restriction disallowing equations between array terms were retained.

Downey and Sethi In [4], Downey and Sethi present a decision procedure for an extensional equational theory with just 1-dimensional arrays. Equations between array terms are allowed. They prove that determining the invalidity of an equation in their theory of arrays is NP-complete.

Nelson and Oppen In [10], Nelson and Oppen describe an extensional theory of arrays. Their theory allows multi-dimensional arrays. They do not present their satisfiability procedure for the extensional theory, but in [9], Nelson gives a detailed presentation of a satisfiability procedure for a non-extensional theory.

Levitt In Chapter 5 of his PhD thesis [7], Levitt presents a decision procedure for an extensional theory of arrays based on solving equations and canonizing terms, in the style of Shostak [13]. A detailed proof of correctness is not given, and has proved elusive to the authors. In contrast, a detailed proof of correctness is given below for the procedure for **Arr**.

3. The satisfiability procedure for Arr

Arr is decided by a refutation procedure. The procedure decides satisfiability of conjunctions of literals, which are equations and disequations between terms. Deciding satisfiability of arbitrary boolean combinations of atomic formulas can be reduced to this problem by well-known means. A conjunction of literals whose satisfiability is to be tested will be called a goal. Comma will be used to denote conjunction. Two goals are said to be *equisatisfiable* when one is satisfiable iff the other is.

3.1. Informal overview

The procedure works in two phases. In the first phase, the original goal is transformed into a set of subgoals such that (i) no subgoal contains *write* and (ii) the original goal is satisfiable iff one of the subgoals is. Eliminating write expressions is straightforward except when they occur as the left or right hand side of an equation. How to eliminate such occurrences of write expressions is the crucial insight of this algorithm.

Definition 2 $(=_{_})$

$$a =_{\mathcal{I}} b \quad \Leftrightarrow_{def} \quad \forall i : I \,.\, i \notin \mathcal{I} \rightarrow read(a, i) = read(b, i)$$

Formulas of the form $a =_{\mathcal{I}} b$ with $\mathcal{I} \neq \emptyset$ are called partial equations.

The crucial observation is that

$$write(a, i, v) = b \;\Leftrightarrow\; (a =_{\{i\}} b \wedge read(b, i) = v).$$

write expressions occurring as sides of equations may thus be eliminated by introducing partial equations.

The second phase of the procedure is based on the observation that in the absence of *write*, arrays behave like uninterpreted functions and *read* behaves like function application. So in the absence of *write*, a congruence closure algorithm (cf. [1]) could be used to decide the theory. The algorithm must be modified to work with partial equations as well as equations, but this can be done. For simplicity, the very simple congruence closure algorithm described in [14] is used, but it should be possible to modify a more complex algorithm.

3.2. Formal presentation

Figure 1 presents our procedure as a proof system. The proof system determines a non-deterministic procedure, where rules are applied bottom-up to analyze a goal into one or more subgoals. The system may be thought of as a rewrite system, where, for each rule, the goal below the line is rewritten to the subgoals above the line. The system resembles a Gentzen-Schütte system where only left rules of the corresponding sequent system are used (i.e., a sequent system where sequents are restricted to be of the form $\Gamma \Rightarrow \bot$). The derivable objects of this system are sets of literals. It is intended that a set of literals be derivable iff their conjunction is unsatisfiable. A *deduction* of a goal is a tree obtained by applying the proof rules bottom-up to that goal. A goal to which no rule can be applied is said to be *normal*.

Phase 1:

(ext) $$\frac{\Gamma,\ \mathit{read}(a,k) \neq \mathit{read}(b,k)}{\Gamma,\ a \neq b}$$ k is not free in the conclusion; a and b are arrays

(r-over-w) $$\frac{\Gamma[v],\ i = j \quad \Gamma[\mathit{read}(a,j)],\ i \neq j}{\Gamma[\mathit{read}(\mathit{write}(a,i,v),j)]}$$

(w-elim) $$\frac{\Gamma,\ a =_{\mathcal{I}} b,\ i \in \mathcal{I} \quad \Gamma,\ a =_{i,\mathcal{I}} b,\ \mathit{read}(b,i) = v,\ i \notin \mathcal{I}}{\Gamma,\ \mathit{write}(a,i,v) =_{\mathcal{I}} b}$$

(w-elim-helper) $$\frac{\Gamma,\ b =_{\mathcal{I}} a}{\Gamma,\ a =_{\mathcal{I}} b}$$ b is a write expression, and a is not

Phase 2:

(partial-eq) $$\frac{\Gamma,\ a =_{\mathcal{I}} b,\ \mathit{read}(a,i) = \mathit{read}(b,i),\ i \notin \mathcal{I} \quad \Gamma,\ a =_{\mathcal{I}} b,\ i \in \mathcal{I}}{\Gamma,\ a =_{\mathcal{I}} b}$$ where $a \succeq b$; $\mathcal{I} \neq \emptyset$; $\mathit{read}(a,i)$ occurs in Γ

(trans) $$\frac{\Gamma,\ a =_{\mathcal{I}} b,\ a =_{\mathcal{I}'} c,\ b =_{\mathcal{I} \cup \mathcal{I}'} c}{\Gamma,\ a =_{\mathcal{I}} b,\ a =_{\mathcal{I}'} c}$$ $\mathcal{I} \neq \emptyset$ and $\mathcal{I}' \neq \emptyset$

(subst) $$\frac{\Gamma[y],\ x = y}{\Gamma[x],\ x = y}$$ $x \succeq y$, $x \not\equiv y$, x not in $\Gamma[\,]$

(symm) $$\frac{\Gamma,\ y =_{\mathcal{I}} x}{\Gamma,\ x =_{\mathcal{I}} y}$$ $x \prec y$

Both phases:

($\in$-split) $$\frac{\Gamma,\ i = j \quad \Gamma,\ i \in \mathcal{I}}{\Gamma,\ i \in (j,\mathcal{I})}$$ ($\notin$-expand) $$\frac{\Gamma,\ i \notin \mathcal{I},\ i \neq j}{\Gamma,\ i \notin (j,\mathcal{I})}$$

($\in$-empty) $$\frac{}{\Gamma,\ i \in \emptyset}$$ (ax) $$\frac{}{\Gamma,\ x \neq x}$$

Figure 1. The decision procedure as a proof system

The system has two phases. Some rules may be applied in just one phase, while others may be applied in either phase. The rules of phase 1 are applied to a goal until no rule applies, and then the rules of phase 2 are applied. The procedure stops and reports that the original conjunction is satisfiable if it encounters a normal subgoal. Otherwise, it reports that the original goal is unsatisfiable. As mentioned before, phase 2 is a modified congruence closure algorithm. The core congruence closure algorithm consists of just the rules (symm) and (subst) [14].

The set-theoretic operators have their usual meanings; note that $i, \mathcal{I}$ denotes $\{i\} \cup \mathcal{I}$, where $\mathcal{I}$ does not contain i. $\Gamma[\,]$ denotes a *context*, which is an expression containing one or more occurrences of a single free variable. The expression obtained by substituting the term t for the context's free variable is written $\Gamma[t]$. In the rule (subst), since the side condition requires that $\Gamma[\,]$ contain no occurrences of the term x, applying (subst) replaces all occurrences of x in $\Gamma[x]$ with the term y. $\equiv$ denotes syntactic identity. The symbol $\preceq$ denotes an ordering on terms by size, which is defined on terms in the usual way. Let $x \preceq y$ iff x and y are such that the size of x is less than or equal to the size of y. The variants $\prec$ and $\succeq$ are derived from $\preceq$ in the usual way.

3.3. Avoiding non-termination in phase 2

In phase 2, applications of (partial-eq) and (trans) must be restricted to avoid certain sources of non-termination. There is nothing preventing (partial-eq) and (trans) from being applied repeatedly with the same partial equations, because for both rules, the partial equations are retained in the goal. For (partial-eq), this form of non-termination may be prevented by adding a side condition to the rule that prevents it from being applied if, informally, $read(a, i)$ and $read(b, i)$ are already known to be equal or if i is already known to be equal to an element of $\mathcal{I}$. Formally, the procedure can test whether or not t and t' are already known to be equal by applying all the rules of phase 2 except (partial-eq) and (trans) to the current goal with $t \neq t'$ added, and seeing whether or not that goal is reported unsatisfiable. If neither ($\in$-split) nor ($\notin$-expand) applies to the current goal, then this is equivalent just to comparing normal forms as determined by the core congruence closure algorithm. So in an implementation, this non-termination may easily be prevented. A similar approach can be used to prevent (trans) from being applied repeatedly to the same formulas. The required machinery, however, has been omitted from the proof system for simplicity.

4. Correctness of the Procedure

A satisfiability procedure is *sound* iff when it reports a goal unsatisfiable, the goal is indeed unsatisfiable. A procedure is *complete* iff when it reports a goal satisfiable, the goal is indeed satisfiable. A procedure is *correct* iff it terminates on all inputs, and it is sound and complete. In this section, a detailed proof of completeness for the satisfiability procedure for **Arr** is given. The proof of termination is routine and omitted for lack of space. The following theorem implies soundness.

Theorem 1 (equisatisfiability) *The conclusion of each rule of the system is satisfiable iff one of its premises is satisfiable.*

Proof: The proof is routine. Consider just the rule (trans). If $a =_{\mathcal{I}} b$ and $a =_{\mathcal{I}'} c$ are true in some model, then it is easy to see by the definition of $=_{_}$ that $b =_{\mathcal{I} \cup \mathcal{I}'} c$ is also true in some model. If c agrees with a at every index except those in $\mathcal{I}'$ and a agrees with b at every index except those in $\mathcal{I}$, then clearly $i \notin \mathcal{I} \cup \mathcal{I}'$ implies that c agrees with a at i and also that a agrees with b at i. Hence, c agrees with b at i. For the other direction, if the premise has a model, so does the conclusion, since the conclusion is a subset of the premise. □

Recall that a normal goal is one to which no rule applies. By the equisatisfiability theorem, to prove completeness of the algorithm it suffices to show that any normal goal is satisfiable. This may be done by constructing a model for a normal goal. The following lemma is easily established.

Lemma 1 (effect of phase 1) *A goal that is normal with respect to phase 1 of the algorithm contains no write expressions and no disequations between array expressions.*

4.1. A convenient form for normal goals

In preparation for constructing a model, several transformations, which are not actually performed by the algorithm, are applied to a normal goal to give an equisatisfiable normal goal Γ, which is in a more convenient form. If the normal goal contains equations of the form $x = x$, clearly they may be removed and the result will be equisatisfiable. Next, modify the goal by doing the following. Let G be the goal as it currently stands. If there is a term of the form $read(a, i)$ in G that is not the left hand side of any equation in G, choose a constant symbol c not occurring in G, and modify G by replacing $read(a, i)$ everywhere in it with c and adding the equation $read(a, i) = c$ to it. If there is no such term $read(a, i)$ in G, stop. It is easy to show that the resulting goal is normal and equisatisfiable with the original normal goal. This resulting goal consists of formulas of one of the following four forms, where x, y, and z are constant symbols:

I. $read(x, y) = z$

II. $x \neq y$

III. $x =_{\mathcal{I}} y$, where every element of $\mathcal{I}$ is a constant symbol

IV. $x = y$

Since this resulting goal is normal, no formula $x = y$ of the form (IV) has its left hand side appearing anywhere else in the goal, since otherwise (subst) would apply. Let Γ be this resulting goal, except without the equations of the form (IV). Γ will be said to be in *convenient normal form*. Any model $\mathcal{M}$ of Γ may be extended to a model of Γ with those equations of the form (IV) by giving the same interpretation for the constant x as for the constant y, if $\mathcal{M}$ interprets y, and a single arbitrary interpretation for both x and y otherwise.

4.2. Construction of a model

In this section, a kind of term model for the goal Γ in convenient normal form is constructed. Several definitions, in terms of Γ, are required. The fact that the core congruence closure algorithm (rules (subst) and (symm)) is correct is used (see [14] for the proof).

Definition 3 ($\rightarrow_$ **and** $\leftarrow_$) *Let $\rightarrow_$ and $\leftarrow_$ be the ternary relations defined, respectively, by*

$$a \rightarrow_{\mathcal{I}} b \quad \textit{iff} \quad (a =_{\mathcal{I}} b) \in \Gamma$$
$$a \leftarrow_{\mathcal{I}} b \quad \textit{iff} \quad (b =_{\mathcal{I}} a) \in \Gamma$$

Note that for any $\mathcal{I}$, $\rightarrow_{\mathcal{I}}$ and $\rightarrow_{\mathcal{I}}$ need not be symmetric, since $(a =_{\mathcal{I}} b) \in \Gamma$ does not imply $(b =_{\mathcal{I}} a) \in \Gamma$.

Definition 4 ($\approx_$) *Let $\approx_$ be the least ternary relation satisfying*

1. $a \approx_{\emptyset} a$, *for every array constant a appearing in Γ*
2. $(a \rightarrow_{\mathcal{I}} b) \vee (b \rightarrow_{\mathcal{I}} a) \rightarrow a \approx_{\mathcal{I}} b$

Definition 5 ($\overset{*}{\approx}_$) *Let $\overset{*}{\approx}_$ be the least ternary relation containing $\approx_$ and satisfying*

$$(\exists c\,.\, a \approx_{\mathcal{I}} c \wedge c \overset{*}{\approx}_{\mathcal{I}'} b) \rightarrow a \overset{*}{\approx}_{\mathcal{I}\cup\mathcal{I}'} b$$

Definition 6 ($\overset{*}{\approx}$) *Let $\overset{*}{\approx}$ be the binary relation defined by*

$$a \overset{*}{\approx} b \quad \textit{iff} \quad \exists \mathcal{I}\,.\, a \overset{*}{\approx}_{\mathcal{I}} b$$

The context will help distinguish $\overset{*}{\approx}_$ and $\overset{*}{\approx}$. Note that $\overset{*}{\approx}$ is an equivalence relation.

Definition 7 (chains) *A chain of applications of a ternary symbol R like $\approx_$ or $\rightarrow_$, called an R-chain, is defined to be a conjunction of the form $(a_1\ R_{\mathcal{I}_1}\ a_2) \wedge (a_2\ R_{\mathcal{I}_2}\ a_3) \wedge \ldots \wedge (a_{n-1}\ R_{\mathcal{I}_{n-1}}\ a_n)$, with $n \geq 2$.*

- *The chain is denoted* $(a_1\ R_{\mathcal{I}_1}\ a_2\ R_{\mathcal{I}_2}\ \ldots\ R_{\mathcal{I}_{n-1}}\ a_n)$.
- *n is the* length *of the chain.*
- *The* union along the chain *is defined to be* $\bigcup_{1\leq j<n} \mathcal{I}_j$.
- *The chain is said to be* from x to y *iff* $a_1 \equiv x$ *and* $a_n \equiv y$.

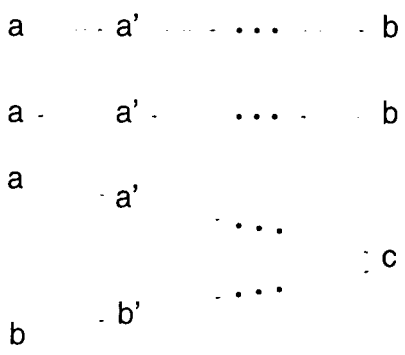


Figure 2. Standard forms for $\approx_$-chains

Lemma 2 (standard form for chains) *Suppose $a \overset{*}{\approx}_{\mathcal{I}} b$, with $\mathcal{I} \neq \emptyset$. Then one of the following is true:*

i. there is a $\rightarrow_$-chain from a to b or from b to a, where the union along the chain is $\mathcal{I}$

ii. for some c, there is a $\rightarrow_$-chain from a to c and another from b to c, where the union of the unions along the two chains is $\mathcal{I}$.

Figure 2 shows the possibilities.

Proof Let C be a $\approx_$-chain $a_1 \approx_{\mathcal{I}_1} \ldots \approx_{\mathcal{I}_{n-1}} a_n$ from a to b, with $\mathcal{I} = \bigcup_{1\leq i\leq n-1} \mathcal{I}_i$. Assume C is of minimal length of all such chains. For every i with $1 \leq i \leq n-1$, let $\leftrightarrow_i$ be either $\rightarrow_{\mathcal{I}_i}$ or $\leftarrow_{\mathcal{I}_i}$, and suppose we have $a_1 \leftrightarrow_i \ldots \leftrightarrow_{n-1} a_n$. It is easy to prove that if this latter chain is not of one of the forms described in (i) and (ii), there must be an i with $1 < i \leq n-1$ such that $\leftrightarrow_{i-1}$ is $\leftarrow_{\mathcal{I}_{i-1}}$ and $\leftrightarrow_i$ is $\rightarrow_{\mathcal{I}_i}$. So we have $a_{i-1} \leftarrow_{\mathcal{I}_{i-1}} a_i \rightarrow_{\mathcal{I}_i} a_{i+1}$. So both $a_i =_{\mathcal{I}_{i-1}} a_{i-1}$ and $a_i =_{\mathcal{I}_i} a_{i+1}$ are in Γ. It must be the case that both $\mathcal{I}_{i-1}$ and $\mathcal{I}_i$ are non-empty, since otherwise (subst) would apply to replace the left hand side of one of those equations by the right hand side of the other. No rules can apply, since Γ is normal. Since both $\mathcal{I}_{i-1}$ and $\mathcal{I}_i$ are non-empty, (trans) would be applicable, unless the conditions described in Section 3.3 for preventing non-termination were keeping it from being applied. This implies that either $a_{i-1} =_{\mathcal{I}_{i-1}\cup\mathcal{I}_i} a_{i+1}$ or $a_{i+1} =_{\mathcal{I}_{i-1}\cup\mathcal{I}_i} a_{i-1}$ is in Γ, since a_1 and a_2 must be their

own normal forms as determined by the core congruence closure algorithm. Hence, we have $a_{i-1} \approx_{\mathcal{I}_{i-1} \cup \mathcal{I}_i} a_{i+1}$. So the chain $a_1 \approx_{\mathcal{I}_1} \ldots a_{i-1} \approx_{\mathcal{I}_{i-1} \cup \mathcal{I}_i} a_{i+1} \cdots \approx_{\mathcal{I}_{n-1}} a_n$, whose union is $\mathcal{I}$, has smaller length than C. This contradicts the assumption that C is of minimal length of such chains. □

Now an interpretation, given as a function $[\![_]\!]$ from the constant and function symbols of Γ to their interpretations, is defined. $[\![_]\!]$ is defined to map every constant symbol a of basic type I to a itself. $[\![_]\!]$ will map array constants to functions. To satisfy extensionality, functions that give the same value for every input are required to be equal. First let $\perp_C$ be a new symbol not occurring in Γ, for every $\overset{*}{\approx}$-equivalence class C. Define $[\![read]\!]$ to be the operation of function application, except that when it is given $\perp_C$, it may just return $\perp_C$. Intuitively, for an array constant a, $[\![a]\!]$ will be a function mapping all but a finite number of inputs to a default value $\perp_C$. Formally, suppose a is in $\overset{*}{\approx}$-equivalence class C. Define $[\![a]\!]$ to be the function that returns $\perp_C$ for every input, except those assigned values by the following:

Definition 8 (interpretation of array constants)
for every constant symbol b of the same type as a,
for every set $\mathcal{I}$ such that $a \overset{*}{\approx}_{\mathcal{I}} b$,
for every index constant i not appearing in $\mathcal{I}$,
if $read(b,i) = x \in \Gamma$ for some x, then
the value of $[\![a]\!]$ for input $[\![i]\!]$ is defined to be $[\![x]\!]$.

Notice that the body of Definition 8 may specify the value for $[\![a]\!]$ on input i more than once. So for $[\![\,]\!]$ to be well-defined, if the value of $[\![a]\!]$ on input i is specified to be $[\![x_1]\!]$ and $[\![x_2]\!]$, we need $[\![x_1]\!] = [\![x_2]\!]$. So if $a \overset{*}{\approx}_{\mathcal{I}} b$ and $a \overset{*}{\approx}_{\mathcal{I}'} c$ with i not in $\mathcal{I}$ and not in $\mathcal{I}'$, then for $[\![\,]\!]$ to be well-defined, it must be the case that if $read(b,i) = x_1, read(c,i) = x_2 \in \Gamma$, then $[\![x_1]\!] = [\![x_2]\!]$. Since the conditions $a \overset{*}{\approx}_{\mathcal{I}} b$, $a \overset{*}{\approx}_{\mathcal{I}'} c$, i not in $\mathcal{I}$, and i not in $\mathcal{I}'$ together imply $b \overset{*}{\approx}_{\mathcal{I} \cup \mathcal{I}'} c$ and i not in $\mathcal{I} \cup \mathcal{I}'$, the following lemma suffices to prove that $[\![\,]\!]$ is indeed well-defined.

Lemma 3 (well-definedness of $[\![\,]\!]$) *If $a \overset{*}{\approx}_{\mathcal{I}} b$, i not in $\mathcal{I}$, and $read(a,i) = x_1, read(b,i) = x_2 \in \Gamma$, then $x_1 \equiv x_2$.*

The proof of this lemma relies on the following sublemma.

Lemma 4 (certain reads equal along chains) *Suppose $a_1, \ldots, a_n$, and i are such that $a_1 \rightarrow_{\mathcal{I}_1} \ldots \rightarrow_{\mathcal{I}_{n-1}} a_n$ for some $\mathcal{I}_1, \ldots, \mathcal{I}_{n-1}$, where i is not in $\bigcup_{1 \le j \le n-1} \mathcal{I}_j$. Suppose there is a constant x such that $read(a_1, i) = x \in \Gamma$. Then $read(a_n, i) = x \in \Gamma$.*

Proof The proof is by induction on n. The base case is trivial. For the induction case, suppose $read(a_1, i) = x \in \Gamma$. Since Γ is normal, no rules can apply. So we must have $\mathcal{I}_1 \neq \emptyset$, since otherwise (subst) would apply with $a_1 = a_2$ and $read(a_1, i)$. Furthermore, since (partial-eq) cannot apply, it must be the case that the conditions of Section 3.3 for preventing non-termination are what is prohibiting its application with $a_1 =_{\mathcal{I}_1} a_2$ and $read(a,i)$. In particular, it must be the case that $read(a_2, i)$ is already known to be equal to $read(a_1, i)$. The other possibility, namely that i is known to be equal to an element of $\mathcal{I}$, is excluded because i is not in $\mathcal{I}$ by hypothesis, and correctness of the core congruence closure algorithm would require i to appear in $\mathcal{I}$ in a normal goal if i were known to be equal to an element of $\mathcal{I}$. For $read(a_1, i)$ and $read(a_2, i)$ to have the same normal form with respect to the core congruence closure algorithm, we must have $read(a_2, i) = x \in \Gamma$; this follows from the definition of convenient normal form. Now the induction hypothesis may be applied to conclude that $read(a_n, i) = x \in \Gamma$. □

Proof (of Lemma 3) Suppose $a \overset{*}{\approx}_{\mathcal{I}} b$ and suppose $\mathcal{I} \neq \emptyset$. Then by Lemma 2, there is either a $\rightarrow_$-chain from a to b or from b to a, or there is a constant c such that there is a $\rightarrow_$-chain from a to c and another from b to c. By Lemma 4, in the first case either $read(b,i) = x_1 \in \Gamma$ or $read(a,i) = x_2 \in \Gamma$, and in the second, $read(c,i) = x_1, read(c,i) = x_2 \in \Gamma$. Since Γ is normal, for all x, y, and z, $read(x,i) = y, read(x,i) = z \in \Gamma$ implies $y \equiv z$, since otherwise (subst) would apply. So in either case, $x_1 \equiv x_2$. If $\mathcal{I} = \emptyset$, then it must be the case that $a \equiv b$, since $read(a,i)$ and $read(b,i)$ are both in Γ; otherwise, (subst) would apply. But again, $read(a,i) = x, read(a,i) = y \in \Gamma$ implies that $x \equiv y$. □

Lemma 5 (correctness of the constructed model) *The model constructed in the previous section satisfies every formula of the goal Γ in convenient normal form.*

Proof Consider the types (I), (II), and (III) of formulas from the list in section 4.1; recall that goals in convenient normal form consist of formulas of just these types.

Case I: $\boldsymbol{read}(x,y) = z$ Since x is an array constant, $x \approx_\emptyset x$, and so the construction of Definition 8 will assign the value that function $[\![x]\!]$ takes on argument $[\![y]\!]$ to be $[\![z]\!]$. Hence $[\![read(x,y)]\!] = [\![z]\!]$.

Case II: $x \neq y$ Since all disequations in Γ are between index expressions, x and y must be index constants. Hence, $[\![x]\!] = x$ and $[\![y]\!] = y$, by construction. If $x \equiv y$, then the goal would not be normal, because (ax) would apply. So the interpretation satisfies $x \neq y$.

Case III: $x =_{\mathcal{I}} y$ It must be shown that for every index constant not in $[\![\mathcal{I}]\!]$, $[\![x]\!]$ and $[\![y]\!]$ give the same value. $[\![x]\!]$ and $[\![y]\!]$ have the same default value since they are in the same $\overset{*}{\approx}$-equivalence class. For those index constants i not

in $\mathcal{I}$ that appear in a formula of the form $read(y, i) = z \in \Gamma$, they store the same values, by Definition 8. □

From the fact that a model has been constructed for a normal goal, the main result now follows.

Theorem 2 (completeness) *The satisfiability procedure for* **Arr** *is complete.*

5. Complexity analysis

Observe that each application of (w-elim) or (partial-eq) leads to one new subgoal for each element of the indexing set $\mathcal{I}$ in the rule. The size of $\mathcal{I}$ is easily seen to be bounded by the size N of the original goal Γ. So any deduction from Γ may be viewed as a tree with branching factor no more than N. It is not hard to show, in fact, that N is an upper bound on the number of branching nodes in the tree, so there are at most $O(N^N) = O(2^{N \lg N})$ branches. Each branch can be shown to be of polynomial length, so the algorithm runs in worst-case exponential time.

Theorem 3 (NP-completeness) *The problem of testing a conjunction of literals for satisfiability in* **Arr** *is NP-complete.*

Proof Downey and Sethi showed that a subproblem of the problem decided here is NP-hard [4]. To show that the problem is in NP, observe that the size of the model constructed in the previous section for a goal Γ in convenient normal form is polynomial in the size of Γ. The conversion of a normal goal to convenient normal form incurs at most a polynomial expansion of the goal. So the size of the model constructed is polynomial in the size of the normal goal. Hence a model can be nondeterministically guessed in polynomial time. Checking whether or not a conjunction of literals is satisfied by a model can be done deterministically in polynomial time. So satisfiability of a conjunction of literals can be checked nondeterministically in polynomial time. □

6. Extensions

In this section, several extensions to the refutation procedure for **Arr** are considered. Due to lack of space, correctness proofs are omitted.

6.1. Propagating all entailed equations

Full incorporation of the satisfiability procedure into the framework for cooperating procedures of [2] requires that the procedure can discover all equations between terms occurring in a satisfiable goal that are entailed by that goal. The procedure for **Arr** always does this for index terms but not always for array terms. If the rules of Figure 3 are added to phase 2, however, it can be shown that if t and t' are array terms in a normal goal that are entailed to be equal, then $t \overset{*}{\approx}_{\emptyset} t'$.

$$\text{(trans2)} \quad \frac{\Gamma,\ a =_{\mathcal{I}} b,\ b =_{\mathcal{I}'} c,\ a =_{\mathcal{I}\cup\mathcal{I}'} c}{\Gamma,\ a =_{\mathcal{I}} b,\ b =_{\mathcal{I}'} c} \quad \text{where } \mathcal{I} \neq \emptyset \text{ and } \mathcal{I}' \neq \emptyset$$

$$\text{(patch)} \quad \frac{\Gamma,\ \neg\phi,\ a =_{i,\mathcal{I}} b \qquad \Gamma,\ \phi,\ a =_{\mathcal{I}} b}{\Gamma,\ a =_{i,\mathcal{I}} b} \quad \text{where } \phi \text{ is } read(a, i) = read(b, i)$$

Figure 3. Rules to propagate entailed equations

6.2. Propagating properly entailed disjunctions

Definition 9 (proper entailment of disjunctions) *A disjunction that is entailed when neither of its disjuncts is entailed is said to be* properly entailed.

Incorporating the procedure into the framework of [2] also requires it to have the following property. Let ϕ and ψ be equations whose sides appear in goal Γ. If the procedure reports Γ satisfiable, then Γ cannot properly entail $\phi \vee \psi$. The original procedure for **Arr** does not have this property; an example is the normal goal $a =_{\{i\}} b, a =_{\{j\}} b, read(b, i) = v, read(b, j) = v'$, which entails $i = j \vee a = b$ but neither $i = j$ nor $a = b$. It can be proved, however, that the modified procedure of section 6.1 does have this property.

6.3. Allowing constant arrays

Constant arrays are arrays that store a single value for all indices. The language is extended with function symbols $const_\tau$ for each value sort τ, and the following axiom schema is added:

$$\forall x : \tau\ .\ \forall i : I\ .\ read(const(x), i) \quad = \quad x$$

The procedure of section 6.1 is modified to obtain a procedure for this extended theory by adding the rules of Figure 4. (const-elim1) is added to both phases, and (const-symm) and (const-elim2) are added to phase 2. To ensure that the conclusion of (const-elim2) entails its premise, the simplifying assumption is made that the interpretation of the type I of indices is infinite. With this modified procedure, goals that are normal with respect to phase 2 may fail to be normal with respect to phase 1. For example, the applications of *const* in the goal $const(write(a, i, v)) = const(b)$

are removed using (const-elim2) in phase 2, but this adds the equation $write(a, i, v) = b$ to the goal, which could be analyzed with the (w-elim) rule of phase 1. So it is necessary to repeat the phases.

$$\text{(const-elim1)} \quad \frac{\Gamma[x]}{\Gamma[read(const(x), i)]}$$

$$\text{(const-symm)} \quad \frac{\Gamma,\ a =_{\mathcal{I}} const(x)}{\Gamma,\ const(x) =_{\mathcal{I}} a}$$

where a is not of the form $const(y)$

$$\text{(const-elim2)} \quad \frac{\Gamma,\ x = y}{\Gamma,\ const(x) =_{\mathcal{I}} const(y)}$$

Figure 4. Rules to treat constant arrays

7. Conclusion

A refutation procedure for an extensional theory of multi-dimensional arrays has been presented and proved correct. The theory **Arr** decided essentially subsumes all previously decided array theories. The procedure is suitable for incorporation into a framework for cooperating decision procedures.

8. Acknowledgements

We thank the anonymous reviewers for their very helpful criticism. The first author was supported during part of this work by a National Science Foundation Graduate Fellowship. Support was also provided in part by NSF contract CCR-9806889-002 and ARPA/AirForce contract F33615-00-C-1693. This paper does not necessarily reflect the position or the policy of the U.S. Government; no official endorsement should be inferred.

References

[1] L. Bachmair and A. Tiwari. Abstract Congruence Closure and Specializations. In D. McAllester, editor, *17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78. Springer-Verlag, 2000.

[2] C. Barrett, D. Dill, and A. Stump. A Framework for Cooperating Decision Procedures. In D. McAllester, editor, *17th International Conference on Computer Aided Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 79–97. Springer-Verlag, 2000.

[3] G. Collins and D. Syme. A Theory of Finite Maps. In *Conference on Higher Order Logic Theorem Proving and its Applications*, 1995.

[4] P. Downey and R. Sethi. Assignment Commands with Array References. *Journal of the ACM*, 25(4):652–666, Oct. 1978.

[5] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[6] D. Kaplan. Some Completeness Results in the Mathematical Theory of Computation. *Journal of the ACM*, 15(1):124–34, Jan. 1968.

[7] J. Levitt. *Formal Verification Techniques for Digital Systems*. PhD thesis, Stanford University, 1999.

[8] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress 62*, 1962.

[9] G. Nelson. Techniques for Program Verification. Technical Report CSL-81-10, Xerox PARC, June 1981.

[10] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[11] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[12] H. Ruess. Private communication. 2000.

[13] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[14] A. Stump, D. Dill, J. Giesl, and C. Barrett. On a Very Simple Abstract Higher-Order Congruence Closure Algorithm. *In preparation*, 2000. Available from http://verify.stanford.edu/~stump/.

[15] N. Suzuki and D. Jefferson. Verification Decidability of Presburger Array Programs. In *Proceedings of a Conference on Theoretical Computer Science*, 1977. University of Waterloo, Waterloo, Ontario, Canada.

On Ordering Constraints for Deduction with Built-In Abelian Semigroups, Monoids and Groups*

Guillem Godoy and Robert Nieuwenhuis
Dept. LSI, Technical University of Catalonia,
Jordi Girona 1, 08034 Barcelona, Spain
E-mail: {ggodoy,roberto}@lsi.upc.es

Abstract

It is crucial for the performance of ordered resolution or paramodulation-based deduction systems that they incorporate specialized techniques to work efficiently with standard algebraic theories E.

Essential ingredients for this purpose are term orderings that are E-compatible, *for the given E, and algorithms deciding constraint satisfiability for such orderings.*

Here we introduce a uniform technique providing the first such algorithms for some orderings for abelian semigroups, abelian monoids and abelian groups, which we believe will lead to reasonably efficient techniques for practice.

The algorithms are optimal since we show that, for any *well-founded E-compatible ordering for these E, the constraint satisfiability problem is NP-hard even for conjunctions of inequations, and our algorithms are in NP.*

Keywords: *symbolic constraints, term orderings, automated deduction.*

1 Introduction

It is crucial for the performance of ordered resolution or paramodulation-based deduction systems that they incorporate specialized techniques to work efficiently with standard algebraic theories E, like abelian semigroups (AC, for associative and commutative) abelian monoids (AC0), or abelian groups (AG).

Essential ingredients for this purpose are *reduction* (i.e., well-founded and monotonic) orderings $\succ$ on ground terms that are *E-compatible* for the given E, i.e., $s =_E s' \succ t' =_E t$ implies $s \succ t$, and algorithms deciding the satisfiability of *ordering constraints* for such orderings. Such ordering constraints are used to express ordered strategies in automated deduction at the formula level [8]. This allows one to reduce the search space by inheriting the ordering restrictions while keeping completeness [13, 15].

An ordering constraint is a quantifier-free first-order formula built over terms in $T(\mathcal{F}, \mathcal{X})$ and over the binary predicate symbols '$=$' and '$>$'. These constraints are interpreted over the domain of ground terms, where $=$ and $>$ are interpreted, respectively, as a congruence $\approx$ and a reduction ordering $\succ$ such that $\succ$ is total up to $\approx$, i.e., for all ground terms s and t either $s \succ t$ or $t \succ s$ or $t \approx s$. Hence a *solution* of a constraint C is a substitution σ with range $T(\mathcal{F})$ and whose domain is the set of variables of C such that $C\sigma$ evaluates to true when interpreting $=$ as $\approx$ and $>$ as $\succ$. Then we say that σ *satisfies* C.

The first practical applications of ordering constraints gave rise to the distinction between *fixed signature* semantics (solutions are built over a given signature $\mathcal{F}$), and *extended signature* semantics (new symbols are allowed to appear in solutions). The latter semantics is in some cases easier to check, and is used in applications like the computation of saturated sets of ordering constrained clauses that can be used for deduction with other clauses containing arbitrary new (e.g., Skolem) symbols, but it is less restrictive and hence less powerful for refutational theorem proving. The satisfiability problem for ordering constraints was first shown decidable for the well-known *recursive path orderings* (RPO) introduced by N. Dershowitz [4], for fixed signatures [2, 7] and extended ones [13, 12]. NP algorithms (fixed and extended signatures) were given in [12, 11]. For the Knuth-Bendix ordering (KBO) this result has only recently been obtained (for fixed signatures) in [9].

Ordered strategies and ordering constraint inheritance can be used without loosing completeness with built-in algebraic theories E, like AC [14, 18] or AG [6]. An additional advantage of constraints in this context is that in

*Both authors are partially supported by the ESPRIT Basic Research Action CCL-II, ref. WG # 22457, and the Spanish CICYT project HEMOSS ref. TIC98-0949-C02-01. The first author is supported by Departament d'Universitats, Recerca i Societat de la Informació de la Generalitat de Catalunya. A version of this paper with all proofs is available from www.lsi.upc.es/~roberto.

0-7695-1281-X/01 $10.00 © 2001 IEEE

each inference only one conclusion is generated, instead of one conclusion for each E-unifier. This can have dramatic consequences. For example, there are more than a million unifiers in $mgu_{AC}(f(x,x,x), f(y_1,y_2,y_3,y_4))$. But, probably due to the lack of adequate orderings and constraint solving algorithms, these ideas have not been put into practice yet. For example, McCune found his well-known AC-paramodulation proof of the Robbins conjecture [10] by still computing complete sets of AC-unifiers, and adding one new equation for each one of them (although heuristics were used to discard some of the unifiers).

Indeed, of the many, rather complex, AC-compatible reduction orderings that have been defined in the literature, only for the AC-RPO ordering of [16] a constraint solving algorithm exists [3]. But, unfortunately, this algorithm is far from practical due to its conceptual and computational complexity, and moreover, it only deals with extended signature semantics.

However, in many practical cases one has to deal with only one single associative and commutative symbol, and then a simple version of the RPO on *flattened* terms, which we will call FRPO, fulfills all requirements. The same FRPO can be used as an ingredient for an AG-compatible reduction ordering AG-RPO that satisfies all requirements of [6], by using it to compare AG-normal forms of ground terms. Finally, it turns out that an AC0-compatible ordering AC0-RPO is obtained in a similar way by considering normal forms w.r.t. the rule $x + 0 \to x$.

Here we introduce a uniform technique providing the first constraint solving algorithms for fixed signature semantics for AC compatible orderings. More precisely, we give NP algorithms for FRPO-based orderings for abelian semigroups, abelian monoids and abelian groups. We believe that the new techniques will lead to reasonably efficient practical algorithms for these orderings, and give new insights for the development of constraint solving methods over fixed signatures for other E-compatible orderings.

This paper is structured as follows. After the basic definitions of Section 2, in Section 3 we deal with FRPO constraints. For explanation purposes, we start with constraints built with a single unary symbol f, a constant symbol 0 and the AC symbol $+$, and later extend it to arbitrary signatures. After explaining the relatively simple extension to AC0-RPO in Section 4, in Section 5 we deal with the hardest part of the paper, namely the techniques for AG-RPO.

It is obvious that the satisfiability problems we deal with are NP-hard, because as subcases they include the AC, AC0 and AG-unifiability problems which are all NP-hard. As a consequence, since our algorithms are in NP, they are optimal, and the problems are NP-complete. But one may wonder whether there exists any ordering at all for these E such that at least the satisfiability problem for positive conjunctions of inequations (by which one cannot always encode unification) is in P. In Section 6, we answer this question negatively: we show that for *any* well-founded total E-compatible ordering for each one of these E, the problem is NP-hard even for conjunctions of positive inequations.

Finally, in Section 7 we give some conclusions and directions for further work.

2 Basic Definitions

We use the standard notation and terminology for terms and constraints of [5] and [15]. The rewrite system R_{AG} consists of the following five rules:

$$\begin{array}{rcl} x+0 & \to & x \\ -x+x & \to & 0 \\ -(-x) & \to & x \\ -0 & \to & 0 \\ -(x+y) & \to & (-x)+(-y) \end{array}$$

By AG we denote the set of seven equations consisting of these five rules (seen as equations) plus AC, the associativity and commutativity axioms for $+$. By AC0 we mean AC $\cup\ R_0 = \{x + 0 \to x\}$. By $=_E$ we denote the congruence on terms generated by a set of equations E. In this paper, rewriting with a set of rules R is always considered *modulo* AC, that is, when writing $\to_{R_{AG}}$, we mean the (convergent) relation $=_{AC} \to_{R_{AG}} =_{AC}$, and terms will always be considered in *flattened form* w.r.t. AC: we consider e.g. $+(a,b,c)$ instead of $+(a,+(b,c))$. Furthermore, $+$ is written in infix notation: $a + b + c$.

Let us first recall the definition of RPO, which allows for variadic symbols (hence we can cope with flattened terms). We assume given a *precedence* $>$ on $\mathcal{F}$, and, for each $f \in \mathcal{F}$, a *status* which is either *multiset* or *lexicographic*. In the following, a symbol will have the multiset status if, and only if, it is variadic. Below, the relation $=^{\text{mul}}$ has to be understood modulo permutations of the direct subterms of any symbol whose status is multiset. More precisely, for every permutation π, if status(f) = multiset, then, for all terms $t_1, \ldots, t_n$, $f(t_1, \ldots, t_n) =^{\text{mul}} f(t_{\pi(1)}, \ldots t_{\pi(n)})$. Then RPO is defined as follows: $s = f(s_1, \ldots, s_n) \succ_{rpo} g(t_1, \ldots, t_m) = t$ iff

1. $\exists\ i \in \{1, \ldots, n\}\ s_i \succ_{rpo} t$ or $s_i =^{\text{mul}} t$, or
2. $f > g$ and $s \succ_{rpo} t_i$ for all $i = 1, \ldots, m$ or
3. $f = g$ and status(f) = multiset, and $\{s_1, \ldots, s_n\} \succ^{\text{mul}}_{\text{rpo}} \{t_1, \ldots, t_m\}$ where $\succ^{\text{mul}}_{\text{rpo}}$ is the multiset extension of $\succ_{rpo}$ or
4. $f = g$ and status(f) = lexicographic, and $(s_1, \ldots, s_n) \succ^{\text{lex}}_{\text{rpo}} (t_1, \ldots, t_n)$ where $\succ^{\text{lex}}_{\text{rpo}}$ is the lexicographic extension of $\succ_{rpo}$.

In the following, we call the RPO on flattened terms FRPO: we define $s \succ_{frpo} t$ if $flat(s) \succ_{rpo} flat(t)$. FRPO is not monotonic in general:

Example 1 *If* $+ > a > b$ *then* $b + b \succ_{frpo} a$ *but* $a + a \succ_{frpo} b + b + a$. *Also, if* $a > + > f$ *then* $f(a) + f(a) \succ_{frpo} f(f(a))$ *but* $f(a) + f(f(a)) \succ_{frpo} f(a) + f(a) + f(a)$. *Similar non-monotonicities occur if there is more than one AC symbol.* □

However, we have the following result:

Lemma 2 ([1]) *If* $+$ *is the only AC symbol and either* $+$ *is the smallest symbol in the precedence, or else only the smallest constant is smaller than* $+$, *then FRPO is an AC-compatible reduction (i.e., monotonic and well-founded) ordering on ground terms that is total up to* $=_{AC}$.

Let us now define the AC0-RPO and AG-RPO orderings. Given two ground terms s and t, we define

$s \succ_{ac0-rpo} t$ if $nf_{R_0}(s) \succ_{frpo} nf_{R_0}(t)$ and

$s \succ_{ag-rpo} t$ if $nf_{R_{AG}}(s) \succ_{frpo} nf_{R_{AG}}(t)$

where $nf_R(s)$ denotes the normal form w.r.t. R of s.

The following is not difficult to prove (see also [6]):

Lemma 3 *AC0-RPO (AG-RPO) is a total AC0-compatible (AG-compatible) reduction ordering on ground terms in normal form w.r.t.* $\rightarrow_{R_0}$ ($\rightarrow_{R_{AG}}$) *if* $+$ *is the only AC symbol and the precedence is of the form* $\ldots > + > 0$ ($\ldots > - > + > 0$).

In the following, we will consider these precedences.

3 FRPO Constraint Solving

For explanation purposes, we present here the simple subcase where the signature contains only $+$, 0, and a unary function symbol f, with the precedence $f > + > 0$.

Let C be an ordering constraint built over f, $+$ and 0, and let T_C be the set of all (sub)terms of C that are: variables, sides of relations $>$ or $=$ in C, terms headed with f, or terms t such that $f(t)$ occurs in C. A *linear constraint* for C is a constraint S of the form

$$t_{1,1} = \ldots = t_{1,k_1} > \ldots > t_{n,1} = \ldots = t_{n,k_n}$$

where all $t_{i,j}$ are distinct and

$\{t_{1,1}, \ldots, t_{1,k_1}, \ldots, t_{n,1}, \ldots, t_{n,k_n}\} = T_C \cup \{0\}$.

We denote by $=_S$ the equivalence relation generated by the equalities in S and by $>_S$ the smallest strict ordering relation on $T(\mathcal{F}, \mathcal{X})$ compatible with $=_S$ and containing the inequalities of S.

Each constraint C can be expressed as an equivalent (i.e., with the same solutions) finite disjunction of linear constraints S for C (see below); similarly, in what follows we will also make the following assumptions:

A1. W.l.o.g. we can assume S to be of the form

$$x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_n = t_{n,1} = \ldots = t_{n,k_n}$$

where $\{x_1, \ldots, x_n\} = \text{vars}(S)$ and all $t_{i,j}$ are distinct non-variable terms. Indeed it is sufficient to insert a new (existentially quantified) variable in each equivalence class without any variables, or to merge two equal variables into one if necessary (merging of equal variables, which will be done more often in this paper, can be recorded separately if one wants to reconstruct a solution for the original constraint rather than to decide its satisfiability).

A2. W.l.o.g. we may assume that each $t_{i,j}$ is either: a sum of variables, or the term 0, or of the form $f(x)$ where x is a variable. This is accomplished by replacing non-variable arguments t by the variable x with $x =_S t$.

A3. W.l.o.g. we may also assume that in each equivalence class $x_i = t_{i,1} = \ldots = t_{i,k_i}$ either all $t_{i,j}$ are headed by $+$ or else the class is simply $x_i = f(x)$ or $x_i = 0$ or x_i. This is the case since equalities between terms headed with different top symbols are trivially unsatisfiable, and linear constraints (to which the previous transformations have been applied) containing equalities $f(x) = f(y)$ are satisfiable only if x and y are the same variable. The rightmost equivalence class can be assumed to be $x_n = 0$: otherwise S is trivially unsatisfiable.

A4. Again w.l.o.g., for comodity of explanations, S can be assumed to be of the form $x = f(z) > \ldots$. A constraint $x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots$ can be transformed, by adding an additional leftmost equivalence class, into $x_0 = f(x_1) > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots$

A5. Every variable x occurring as a proper subterm in S can w.l.o.g. be assumed to have another occurrence to the right of it in S at top level (i.e., not as a proper subterm of another term). Otherwise, S is trivially unsatisfiable.

A6. One may assume that if $f(x) >_S f(y)$, then also $x >_S y$. Otherwise, S is again trivially unsatisfiable.

A7. If we have $y_1 + \ldots + y_k \geq_S f(y)$, then, for some i in $1 \ldots k$ we have $y_i \geq_S f(y)$. Otherwise, S is again trivially unsatisfiable.

Example 4 *Let the constraint* C *be* $f(x + z) > y \wedge z > f(x)$. *One of its linear constraints is* $y = f(x + z) > f(x) > x + z > x = z = 0$. *Enforcing the assumptions, it becomes* $y = f(w_2) > w_1 = f(x) > w_2 = x + x > x = 0$ *by adding new variables* w_1 *and* w_2 *for the classes of* $f(x)$ *and* $x + z$ *respectively, and merging* x *and* z. *However, it*

is in contradiction with our initial constraint C. Another linear constraint is $f(x+z) > x+z > z = y > f(x) > x = 0$, *which becomes* $w_1 = f(w_2) > w_2 = x+y > y > w_3 = f(x) > x = 0$. *This linear system satisfies all our assumptions and it is not in contradiction with C.* □

Lemma 5 ([2, 12]) *Each constraint C can be transformed into a finite disjunction of linear constraints satisfying the previous assumptions, and such that C is satisfiable if and only if one of the linear constraints is.*

3.1 Segments and the splitting transformation

A term u is a *summand* if it is headed with a symbol different from $+$. It is a *top-level summand* of a term t if t is of the form u or $u+t'$. A *segment* T of a linear constraint S is a subsequence of S of the form

$x_0 = f(s) > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t$

where t is 0 or headed with f and all $t_{i,j}$ are sums of variables. The variables $x_1, \ldots, x_{i+1}$ are said to be the *defined variables* of T, and their occurrences as single variables in their equivalence classes are their *definitions*.

In such a segment T, every variable occurring in some $t_{i,j}$ is defined either in T itself or in some other segment to the right of T. Now our aim is to transform S in such a way that the latter kind of variables are removed from T, while preserving satisfiability. On the other hand, as a result of this transformation, terms $f(v)$ where v is a sum of variables may appear in S.

The idea is as follows. Let σ be some arbitrary solution of S, let x be a variable defined in T, and let y be the variable defined in the equivalence class immediately below x, that is, x is x_j with $1 \leq j \leq i$, and y is x_{j+1}. Then $x\sigma \succ y\sigma \succeq t\sigma$. Therefore, for at least one of the top-level summands u of $x\sigma$ we have $u \succeq t\sigma$. Hence, if U_x is the sum of all top-level summands u of $x\sigma$ with $u \succeq t\sigma$, and u_x is the (possibly empty) sum of the smaller ones, then $x\sigma$ is of the form $U_x + u_x$ or of the form U_x. Similarly, $y\sigma$ can be of the form $U_y + u_y$ or U_y. Furthermore, either (i) $U_x \succ U_y$, or else, if u_x is non-empty, (ii) $U_x = U_y$ and u_y is empty or $u_x \succ u_y$. In the former case, we say that $x > y$ due to the "large" sumands, and in the latter case due to the "small" summands.

According to these ideas, S will be transformed by the following *splitting transformation*, treating one whole segment T at the same time, segment by segment from left to right, except for the rightmost segment, that does not need any treatment. One can assume that in segments T' to the left of T, all variables not below f are defined in T'. Let T be:

$x_0 = f(s) > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t$

1. Guess a subset of *split* variables of $\{x_1 \ldots x_i\}$ such that whenever $x =_S y_1 + \ldots + y_k$, then x is split if, and only if, at least one of the y_i is split or defined in a segment to the right of T (intuitively, x is split if it is guessed to have at least one "small" summand).

2. If x is a split variable, then introduce two new variables X and x', and everywhere in S replace x by $X + x'$. In this case we say that x is *split* into $X + x'$ (intuitively, the X is for the large summands and the x' for the small ones). If x is a non-split variable of $\{x_1 \ldots x_{i+1}\}$, replace x by a new variable X.

3. After this, the equivalence classes e in the segment are either of the form $V_1 + v_1 = \ldots = V_k + v_k$ or of the form $V_1 = \ldots = V_k$, where the V_i are sums of upper case variables and the v_i are sums of lower case variables and variables defined in segments to the right of T. If e is such an equivalence class, we denote by E the equivalence class $V_1 = \ldots = V_k$ and by e' the class $v_1 = \ldots = v_k$ (if it exists for e). Then we can write T as $x_0 = f(s) > e_1 > \ldots > e_{i+1}$ and we can guess, for each relation $e_j > e_{j+1}$ whether (i) it is due to the large summands or (ii) to the small ones (note that case (ii) applies only if e'_j is non-empty). Accordingly, replace T by the new segment T':

   $x_0 = f(s) > E_1 \# \ldots \# E_{i+1}$

   Furthermore, insert each e'_j in a segment to the right of T, adding it to an existing equivalence class or creating a new one, in such a way that, whenever $E_j =_{T'} E_{j+1}$, either $e'_j > e'_{j+1}$ or e'_{j+1} does not exist.

This transformation does not increase the number of segments of S and only a polynomial number of variables are split: each variable can only lead to k splittings, where k is the number of segments.

Example 6 *(Example 4 continued) Let us apply the splitting transformation to the result* $w_1 = f(w_2) > w_2 = x+y > y > w_3 = f(x) > x = 0$ *of Example 4. First we treat the leftmost segment* $w_1 = f(w_2) > w_2 = x+y > y > w_3 = f(x)$. *The possible variables to be split are* w_2 *and* y. *We guess to split only* w_2 *into* $W_2 + w'_2$, *obtaining* $w_1 = f(W_2 + w'_2) > W_2 + w'_2 = x+y > y > w_3 = f(x)$. *Now, for the relation* $W_2 + w'_2 > y$ *we guess* $W_2 = y$. *After removing* w'_2 *from this segment and inserting it, for example, in the equivalence class of* 0, *we obtain* $w_1 = f(y+x) > y > w_3 = f(x) > x = 0$. *For the segment* $w_3 = f(x) > x = 0$ *no splitting is needed.* □

Definition 7 *We say that two sums of variables* $X_1 + \ldots + X_k$ *and* $Y_1 + \ldots + Y_l$ *are* compared by segments in S' *if:*

- *For all* i *in* $1 \ldots k-1$ *the segment where* X_{i+1} *is defined is to the right of the segment where* X_i *is defined, and the same for the* Y_i*'s for* i *in* $1 \ldots l-1$.

- *There exists an i in $1 \ldots k$ such that $X_j =_S Y_j$ for all j with $j < i$, and either $i = l+1$ or $i \leq l$ and $X_i >_S Y_i$.*

If $x >_S y$ and S' is obtained from S by the splitting transformation, then the occurrences of $f(x)$ and $f(y)$ in S become $f(X + X' + X'' + \ldots)$ and $f(Y + Y' + Y'' + \ldots)$ in S', respectively, where the sums $X + X' + X'' + \ldots$ and $Y + Y' + Y'' + \ldots$ are compared by segments in S'.

3.2 Diophantine systems

Assume S is the result of applying the splitting transformation to a linear system. Now we can define a system of diophantine equations and inequations D_S for S as follows. For each segment T in S of the form

$x_0 = f(s) > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t$

the system D_S contains the equations and inequations:

1. $x_1 > x_2,\ x_2 > x_3,\ \ldots,\ x_i > x_{i+1}$
2. $x_j = t_{j,k}$, for all j in $\{1 \ldots i\}$, and all k in $\{1 \ldots k_j\}$
3. the equation $x_{i+1} = 1$.

Example 8 *(Example 6 continued) The system of diophantine equations for $w_1 = f(y + x) > y > w_3 = f(x) > x = 0$ is*

$$w_1 = 1 \quad y > w_3 \quad w_3 = 1 \quad x = 1$$

We obtain a solution θ for it by defining $y\theta = 2$. Below we will see that from each such a θ one can build a solution σ for the linear constraint from right to left. We have $x\sigma = 0$ and hence $w_3\sigma = f(0)$. Now for each variable v with $v\theta = n$, we define $v\sigma = t + \ldots^{n)} + t$, where t is the summand at the lower end of its segment; e.g., we define $y\sigma$ to be $f(0) + f(0)$. Finally, we have $w_1\sigma = f(f(0) + f(0) + 0)$. If one desires to reconstruct the solution for the original constraint of Example 4: $w_2'\sigma$ is 0, and $z\sigma$ is $f(0) + f(0)$. □

The following simple result will be used below when solving ordering constraints on multisets of several elements as multisets over a single element:

Lemma 9 *Let C be a set $\{c_n, \ldots, c_0\}$ with an ordering $\succ$ where $c_n \succ \ldots \succ c_0$. Then for any decreasing sequence of finite multisets over C*

$$M_0 \gg \ldots \gg M_m$$

there exists a weighting function $f : C \to \mathcal{N}$ with $f(c_0) = 1$ such that

$$F(M_0) > \ldots > F(M_m)$$

where the extension to multisets F of f is defined $F(\{a_1 \ldots, a_k\}) = f(a_1) + \ldots + f(a_k)$.

Proof: Let k be $n_0 + \ldots + n_m$. Then, for instance, the function $f(c_i) = k^i$ fulfills the requirements. □

Lemma 10 *Let $S_1 \ldots S_m$ be the resulting systems of applying the splitting transformation to a linear constraint S over the signature $f > + > 0$. Then S is satisfiable for FRPO if, and only if, some D_{S_i} is satisfiable in the positive natural numbers.*

Proof: $\Longleftarrow$: Assume $D_{S'}$ is satisfiable for some S' in $\{S_1 \ldots S_m\}$. Let θ be a solution for $D_{S'}$. We can build a solution σ for S' as follows. For each segment T in S of the form

$x_0 = f(s) > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t$

assume a (partial) solution σ has already been defined for all segments to the right of T. Then, for the variables x_j defined in this segment we define $x_j\sigma$ to be $t\sigma + \ldots^{n)} + t\sigma$ where $n = x_j\theta$ (note that if T is the rightmost segment, then t is 0). Clearly, σ satisfies all equality relations in S', that is, $u\sigma =_{\mathrm{AC}} v\sigma$ for all u and v with $u =_{S'} v$. Furthermore, it also satisfies the relations $x_j\sigma \succ x_{j+1}\sigma$ with j in $\{1 \ldots i\}$ for such segments T.

Hence it only remains to be checked that σ satisfies $f(s)\sigma \succ x_1\sigma$. Since $x_1\sigma$ is of the form $t\sigma + \ldots + t\sigma$ and $f > +$, it suffices to check that $f(s)\sigma \succ t\sigma$, where t is headed with f (the case where t is 0 is trivial). Then $f(s)$ is of the form $f(X + X' + X'' + \ldots)$ and t is of the form $f(Y + Y' + Y'' + \ldots)$, as a result of the splitting transformation applied to terms $f(x)$ and $f(y)$.

But by assumption A6, if $f(x) >_S f(y)$, then also $x >_S y$. Therefore, our result follows: after the splitting transformation, the sums $X, X', X'', \ldots$ and $Y, Y', Y'', \ldots$ are compared by segments in S', and σ assigns one different summand to each segment, and in the correct order.

Once we have this solution σ for S', it can be extended to a solution for S by recursively defining $x\sigma$ to be $X\sigma + x'\sigma$, for each splitting of a variable x into $X + x'$.

$\Longrightarrow$: Assume S is satisfiable. Now we prove that $D_{S'}$ is satisfiable as well for some S' in $\{S_1 \ldots S_m\}$. Let σ be a solution of S. Let S' be the system obtained by applying the splitting transformation according to σ, that is, if x is defined in a segment T of S of the form

$x_0 = f(s) > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t,$

then x is split into $X + x'$ if $x\sigma$ contains any summands smaller than $t\sigma$; we proceed similarly for the other guessings, and σ is extended conveniently for the new variables. The extended substitution σ is a solution for S'. Moreover, in a segment of S' like the previous one, for all j in $\{1 \ldots i+1\}$ we have that $x_j\sigma$ contains only top-level summands greater than or equal to $t\sigma$.

Now let $C = \{u_0, \ldots, u_n\}$ be all the different top-level summands of these variables, where $u_n \succ u_{n-1} \succ$

$\ldots \succ u_0$ and u_0 is $t\sigma$. Every $x_j\sigma$ and $t_{j,l}\sigma$ can be seen as a multiset on these summands (the multiset of its top-level summands). By Lemma 9 there exists a function $f : C \to \mathcal{N}$ such that its extension F to multisets satisfies $F(x_1\sigma) \gg \ldots \gg F(x_{i+1}\sigma)$, and $F(x_{i+1}\sigma) = f(u_0) = 1$. Moreover, since $x_j\sigma$ and $t_{j,l}\sigma$ are the same multiset, if $t_{j,l}$ is of the form $x_{j_1} + \ldots + x_{j_l}$, then $F(x_j\sigma) = F(t_{j,l}\sigma) = F(x_{j_1}\sigma) + \ldots + F(x_{j_l}\sigma)$. Therefore, the assignment $x_j = F(x_j\sigma)$ satisfies the equations of $D_{S'}$ corresponding to T. □

Theorem 11 *The satisfiability problem for FRPO constraints over the signature $f > + > 0$ is in NP.*

Proof: Generating one of the linear constraints S of the disjunction equivalent to C consists of a polynomial number of guessings of the relations between all the subterms in C, and the size of S is polynomial w.r.t. the size of C. The splitting transformation consists of a polynomial number of guessings. By Lemma 10 S is satisfiable if and only if there exists a sequence of guessings, in the splitting transformation, giving a linear constraint S', such that $D_{S'}$ is satisfiable. Checking whether $D_{S'}$ is satisfiable is again in NP [17]. □

3.3 More function symbols

We consider now the case where the signature contains any finite number of function symbols with arbitrary arities. The precedence is now of the form $\ldots > + > 0$.

W.l.o.g., the following additional assumptions w.r.t. the linear constraint generated from the initial constraint may be assumed (otherwise the linear constraint is again trivially unsatisfiable):

A8. If $f(x_1, \ldots, x_n) >_S g(y_1, \ldots, y_m)$ and $g > f$, then $x_i \geq_S g(y_1, \ldots, y_m)$ for some i in $1 \ldots n$.

A9. If $f(x_1, \ldots, x_n) >_S f(y_1, \ldots, y_n)$ then either $x_i \geq_S f(y_1, \ldots, y_n)$ for some i in $1 \ldots n$ or else $(x_1, \ldots, x_n) >_S^{lex} (y_1, \ldots, y_n)$.

Segments are defined as before, except that now the function symbols at the begining and at the end of it may be different: a segment T of a linear constraint S is a subsequence of S of the form

$x_0 = s > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t$

where s and t are not headed with $+$ and all $t_{i,j}$ are sums of variables. The splitting transformation and the diophantine system are defined exactly as before.

Lemma 12 *Let $S_1 \ldots S_m$ be the resulting systems of applying the splitting transformation to a linear constraint S. Then S is satisfiable if, and only if, some D_{S_i} is satisfiable.*

Theorem 13 *The satisfiability problem for FRPO constraints is in NP.*

4 AC0-RPO Constraints

In this section we consider AC0-RPO constraints over arbitrary signatures of the form $\ldots > f > + > 0$. Observe that all terms of the form $0 + \ldots + 0$ are equivalent to 0 in this setting and that hence the second smallest term w.r.t. the ordering $\succ$ is $f(0, \ldots, 0)$. Therefore we can add, w.l.o.g., an aditional assumption to our linear constraints:

A10 All linear constraints S are of the form $S' > x = f(y, \ldots, y) > y = 0$ and no term of the form $t + y$ occurs in S.

With this additional assumption, it is easy to see that the whole rest of the steps described in the previous section directly suffice for AC0-RPO constraints. Minor details are that, during the splitting process, the new assumption A10 has to be preserved, and then, no small variables resulting from a splitting can be inserted in the rightmost segment. Moreover, in the diophantine system it is not necessary to create the equations corresponding to the rightmost segment.

Observe that the basic idea of the splitting process is that solutions for the linear constraint are transformed into new solutions where, at every segment, the variables that appear in it contain only top-level summands of this segment. Therefore, 0 does not appear in segments that are not the rightmost one, and hence everything behaves like in the FRPO case, again solving the diophantine equations over the positive natural numbers. This gives us the following result.

Theorem 14 *The satisfiability problem for AC0-RPO constraints is in NP.*

5 AG-RPO Constraints

In this section we consider AG-RPO constraints over arbitrary signatures of the form $\ldots > - > + > 0$. In this context summands are terms headed with some symbol different from 0, $+$ or $-$.

Let us first consider some examples over the signature $f > a > - > + > 0$ where f is unary and a is a constant.

Example 15 *Then the smallest terms over this signature in increasing order w.r.t. $\succ$ are:*

$0,\ a,\ a+a,\ a+a+a,\ \ldots,\ -a,\ -a-a,\ -a-a-a,\ \ldots,\ f(0),\ f(0)+a$

$f(0)+a+a,\ \ldots,\ f(0)-a,\ f(0)-a-a,\ \ldots,\ f(0)+f(0),\ f(0)+f(0)+a,\ \ldots,\ -f(0)$

where $-a$ is the smallest limit ordinal ω, $f(0)$ is 2ω, $f(0)-a$ is 3ω, $f(0)+f(0)$ is 4ω, $-f(0)$ is ω^2, and $f(a)$ is $2\omega^2$. □

Example 16 *We have $f(f(a)) \succ f(a-f(0)+f(a-a))$ since*

$\mathrm{nf}_{R_{AG}}(f(f(a))) = f(f(a)) \succ_{FRPO} f(a) = \mathrm{nf}_{R_{AG}}(f(a-f(0)+f(a-a)))$. □

Example 17 *Terms can be smaller than their subterms: $\sigma \models x > f(x-f(a))$ if $x\sigma = f(a)$, since $\mathrm{nf}_{R_{AG}}(f(a)) = f(a) \succ_{FRPO} f(0) = \mathrm{nf}_{R_{AG}}(f(f(a)-f(a)))$.* □

Since, as we have seen in the previous example, a linear constraint such that x appears to the right of the segment where it is defined may be satisfiable, assumption A5 will not be made in this section. Similarly, the following example shows us that terms headed with f may become equal to terms headed with $+$ or $-$. Hence assumption A3 is also dropped in this section:

Example 18 *$\sigma \models x-y = f(z)$ if we have $x\sigma = f(a)+f(a)$, $y\sigma = f(a)$, $z\sigma = a$.* □

An other difficulty to be taken into account is that, after the splitting transformation, contrarily to what happened in the previous sections, a solution for a linear constraint may need more than one different top-level summand for some segments:

Example 19 *Suppose that we have a signature of the form $f > - > + > 0$ where f is unary. Then the smallest terms are ordered like:*

$0,\ f(0),\ f(0)+f(0),\ f(0)+f(0)+f(0),\ \ldots,\ -f(0),\ -f(0)-f(0),\ -f(0)-f(0)-f(0),\ \ldots,\ f(f(0))$.

The linear constraint $f(f(0)) > -z > z > y > -y > f(0)$ is unsatisfiable: since we need to satisfy $y > -y$, necessarily $y\sigma$ is a sum of negative $f(0)$'s. Therefore $z\sigma$ is of the form $-f(0)-\ldots-f(0)$, with some more negative $f(0)$'s. But then $-z > z$ is not satisfied by σ.

However, the linear constraint $f(f(f(0))) > -z > z > y > -y > f(0)$ has the solution σ where $y\sigma = -f(0)-f(0)$ and $z\sigma = f(f(0))+f(f(0))$. It has no solution where $y\sigma$ and $z\sigma$ are built from one single summand. □

5.1 Only unary symbols

For explanation purposes, in this subsection we first assume that all the non-constant function symbols have arity one. Our signature is of the form $\ldots > h > c_1 > \ldots > c_l > - > + > 0$, where h is the smallest non-constant function symbol, i.e., all the c_i are constants.

Then we have the following ordering on summands (from which the ordering on ground terms is easily derived). If $l = 0$ then the smallest summands are, in increasing order: $h(0)$, $h(h(0))$, $h(h(h(0)))$,... If $l \neq 0$ then the smallest summands are, in increasing order: c_l, ..., c_1, $h(0)$, $h(c_l)$, $h(c_l + c_l)$, $h(c_l+c_l+c_l)$,.... These summands will be denoted by $sum_1, sum_2, sum_3, \ldots$

Note that the successor summand of a summand of the form $h(s)$ is $h(s+sum_1)$ if s is not of the form $s'-sum_1$, and $h(s-sum_1)$ otherwise. The successor summand of a summand $f(s)$ with $f > h$ is always $h(f(s))$. We write $succsum_k(u)$ to denote the k-th successor summand of u.

5.1.1 Conditions for the linear constraints.

As before, we generate a disjunction of linear constraints, and apart from the assumptions $A1 - A9$, except, as said, A3 and A5, we need:

A11. W.l.o.g. one can assume that all the constants c_i and the terms sum_1, sum_2 and $h(0)$ appear in S, and in the correct order. We will refer to the segment between sum_2 and sum_1 as the *base segment*.

A12. Every variable x is defined to the right of all occurrence of the form $f(x)$.

A13. There is no $f(x) =_S g(y)$ for $f \neq g$ or $x \neq_S y$. Therefore we may assume that each equivalence class is of the form $x_i = t_{i,1} = \ldots = t_{i,k_i}$ or $x_i = t_{i,1} = \ldots = t_{i,k_i} = f(x_l)$, where all $t_{i,j}$ are sums of positive and negative variables.

A14. All linear constraints are of the form $S' > x = \ldots = sum_1 > y = \ldots = 0$ and no term of the form $t+y$ occurs in S.

In all assumptions, the symbols f and g refer always to functions different from $+$ and $-$. Conditions A12 and A13 are weaker versions of conditions A5 and A3 respectively. Condition A14 is a modification of condition A10: in the class of 0, sums of variables defined to the left of it may appear; in a solution for the constraint, these variables will contain summands that cancel each other out.

In this setting, a sum of variables is, in fact, a sum of positive and negative variables, and all assumptions have to be interpreted accordingly. For example, condition A7 implies that no term of the form $-x$ is in a segment to the left of the segment where x is defined.

5.1.2 The splitting transformation.

The splitting transformation is essentially as before, with some differences. For example, when we guess that some

relation is due to the small summands, the small terms cannot be inserted in the class of 0. Therefore, it makes no sense to do any splitting of variables in the base segment. Another difference with the previous cases is that after splitting and removing small variables from a segment T, some variables defined in T may appear to the right of T. For this reason, we need to introduce the so-called *associated* equations, a set of equations associated to each segment, but that is not inserted in the linear constraint. During the splitting transformation, just after removing the small variables of a segment T, equations are associated to T as follows. Let s be a term in an equivalence class to the right of T, and suppose that s is of the form $M + m$ or $f(M + m)$, where M is a sum of positive and negative variables defined in T (i.e. upper case variables at this point), and m does not contain any of these variables. Then clearly in any solution σ the term $M\sigma$ must be equivalent to 0. Therefore, for each such s, the part M is removed from s, and $M = 0$ becomes an associated equation of T (if the part m of s is empty, then M is replaced by x, the variable of the class of 0).

Finally, for explanation purposes, we want the rightmost class of each T to be of the form $x = t$, for some term t not headed by $+$ (remember: since condition A3 is dropped, there can be other terms headed with $+$ in this class). This can be accomplished as follows. Assume after splitting, this class is of the form $x = T_i + t'_1 = \ldots = T_l + t'_l = t$, where the T_i are the "large" sums, i.e., the sums of the positive and negative variables defined in T. Then the class $t'_1 = \ldots = t'_l$ necessarily has to be inserted in the class of 0. Furthermore, the T_i's are removed as well, and the equations $x - T_i = 0$ are added as additional associated equations of T.

By processing the segments in this manner, from left to right, when we arrive to the segment containing the class of 0, it is of the form $x = sum_1 > x = 0$, since the rest of variables cannot appear in this segment, at this point.

Example 20 *Let us consider the signature* $h > - > + > 0$. *Suppose during the splitting transformation just after splitting the variables of the leftmost segment we obtain:*

$z = h(x_3) > x_3 > x_2 > x_1 > x_0 = x_3 - x_2 - x_1 + y_2 - y_1 - y_1 = h(y_1) >$

$y_3 = x_2 - x_1 - x_0 + y_2 + y_1 > y_2 > y_1 > h(w) > w = 0$.

At this point, if we assume that this splitting of variables has been done according to a solution σ, *then, all the* $x_i\sigma$ *contain top-level summands bigger than or equal to* $h(y_1)\sigma$, *and all the* $y_i\sigma$ *contain top-level summands smaller than* $h(y_1)\sigma$. *Since* $(x_3 - x_2 - x_1 + y_2 - y_1 - y_1)\sigma$ *must coincide with* $h(y_1)\sigma$, *the summands below the* $y_i\sigma$*'s must cancel each other, i.e.* $(y_2 - y_1 - y_1)\sigma$ *must be* 0. *Therefore, we remove* $y_2 - y_1 - y_1$ *from the sum* $x_3 - x_2 - x_1 + y_2 - y_1 - y_1$, *and add it to the class of* 0, *obtaining:*

$z = h(x_3) > x_3 > x_2 > x_1 > x_0 = x_3 - x_2 - x_1 = h(y_1) >$

$y_3 = x_2 - x_1 - x_0 + y_2 + y_1 > y_2 > y_1 > h(w) > w = y_2 - y_1 - y_1 = 0$

Now, in order to leave the treated segment in a normalized form $x_0 = h(y_1)$, *we remove the* $x_3 - x_2 - x_1$ *and we add* $x_0 - x_3 + x_2 + x_1 = 0$ *to the set of associated equations of this segment.*

Finally, since the term $x_2 - x_1 - x_0 + y_2 + y_1$ *is to the right of* $h(y_1)$, *and hence it must contain only summands smaller than* $h(y_1)\sigma$, *we have to force the* x_i*'s to cancel each other. We remove* $x_2 - x_1 - x_0$ *and we add* $x_2 - x_1 - x_0 = 0$ *to the associated equations of the leftmost segment. Note that this is a different treatment with respect to what was done with* $y_2 - y_1 - y_1$ *before. But remember that the aim is to remove variables of the treated segment from the other segments to the right of it. In fact, this* $y_2 - y_1 - y_1$ *added to the class of* 0 *will be removed from this class when we will treat the next segment, since none of the* y_i*'s is defined in the rightmost segment.*

Just after finishing the treatment of the leftmost segment we obtain:

$z = h(x_3) > x_3 > x_2 > x_1 > x_0 = h(y_1) >$

$y_3 = y_2 + y_1 > y_2 > y_1 > h(w) > w = y_2 - y_1 - y_1 = 0$

where the leftmost segment contain the associated equations $x_0 - x_3 + x_2 + x_1 = 0$ *and* $x_2 - x_1 - x_0 = 0$. □

5.1.3 Diophantine equations.

Example 19 shows that now in solutions more than one summand may be needed in a single segment. But only a certain number of summands play an important role in the comparisons.

Example 21 *If* $a > b > c$, *in the inequation* $a+a+a+b+b+c \succ a+a+a-c-c-c$ *the summand* b *will be called the* decisive *summand, since it is the greatest sumand that appears in both terms with a different number of occurrences.* □

Let s be a term and u a summand. The *number of occurrences* of u in s (notation $\#(u, s)$) is the integer n such that $s =_{AG} nu + s'$, where u is not a top-level summand of s'. For instance $\#(a, f(a+b) - a - a) = -2$. Let s and t be two ground terms such that $s \succ t$. The *decisive summand* of the inequation $s \succ t$ is the top-level summand u such that for all summands $v \succ u$, $\#(v, s) = \#(v, t)$, and either (i) $\#(u, s) > \#(u, t) \geq 0$ or (ii) $\#(u, s) < \#(u, t)$ and $\#(u, s) < 0$.

Once the splitting transformation has been applied to S, we can define a system of diophantine equations and inequations D_S for S as follows. For each segment T in S of the form

$x_0 = s > x_1 = t_{1,1} = \ldots = t_{1,k_1} > \ldots > x_i = t_{i,1} = \ldots = t_{i,k_i} > x_{i+1} = t$

with associated equations $q_1 = 0, \ldots, q_l = 0$, several guessings are necessary. First, the number $ndec$ of the segment

is guessed. Intuitively, for a given σ, the number $ndec$ is the cardinality of $dec(T\sigma) \cup \{t\sigma\}$, where $dec(T\sigma)$ is the set of different decisive summands in the inequations $x_j\sigma \succ x_{j+1}\sigma$ with $j > 0$. Hence one can guess $ndec$ to be between 1 and $i+1$. There are some cases where it must be exactly 1, which is when we know that for all σ we have $s\sigma = succsum_1(t\sigma)$:

- s is some c_j and t is c_{j+1}, or
- t is c_1 and s is $h(0)$, or
- t is sum_1 and s is sum_2, or
- t is headed with some f with $f > h$ and s is $h(x_{i+1})$.

In the following, the elements of $dec(T\sigma) \cup \{t\sigma\}$ are denoted (and ordered) by $u_{ndec} \succ \ldots \succ u_1$. Note that always $t\sigma$ is u_1 (if the splitting has been done according to σ).

Now, for every variable x_j with $1 \leq j \leq i+1$ we create $ndec$ integer variables $x_{j,1}, \ldots, x_{j,ndec}$, representing the number of occurrences of each decisive summand in x_j. For the segments where $ndec$ is 1 (as for the base segment) we preserve the same variable name x_j for the corresponding integer variable.

Example 22 *Consider $f > h > - > + > 0$ and suppose that after the splitting transformation we have:*

$z_4 = h(w_1 + x_2) > w_6 = -w_5 > w_5 > w_4 = -w_3 > w_3 > w_2 = -w_1 > w_1 = f(z_3) >$

$z_3 = h(x_3) > y_4 = -y_3 > y_3 > y_2 = -y_1 > y_1 = h(x_2) >$

$z_2 = h(x_1) > x_3 > x_2 > x_1 = h(z_1) > z_1 = 0$

Now, we want to find a solution σ such that for every variable it contains summands greater than or equal to the rightmost term of the segment where it is defined. We may guess that the number of decisive summands for the leftmost segment is 3. Therefore, we need to guarantee that at least two summands between $f(z_3)\sigma$ and $h(w_1+x_2)\sigma$ exist. Observe that the successor summand of $f(z_3)\sigma$ is $h(f(z_3))\sigma$ and the next one is $h(f(z_3)+h(0))\sigma$. Since x_2 is a variable in the base segment, we need $x_2\sigma$ to be at least $h(0)+h(0)$. Here appears the need of adding, to the diophantine system, either an equation of the form $x_2 \geq 2$ or one of the form $x_2 < 0$, since $-f(0)$ is greater than any sum of positive $f(0)$'s.

Later on, we may decide that the number of decisive summands for the segment $z_3 = h(x_3) > y_4 = -y_3 > y_3 > y_2 = -y_1 > y_1 = h(x_2)$ is 2. We need to guarantee that there exists at least one summand between $h(x_3)\sigma$ and $h(x_2)\sigma$. Observe that x_3 and x_2 are defined in the base segment. If we guess $x_2\sigma$ to be $h(0)+\ldots+h(0)$, then either $x_3\sigma$ is also of the form $h(0)+\ldots+h(0)$ with at least two more $h(0)$'s than $x_2\sigma$, or $x_3\sigma$ is of the form $-h(0)-\ldots-h(0)$. If we guess that $x_2\sigma$ is $-h(0)-\ldots-h(0)$, then $x_3\sigma$ also has to be $-h(0)-\ldots-h(0)$, but with at least two more $-h(0)$'s than $x_2\sigma$. □

We now impose some more diophantine equations ensuring that there will be enough space for the decisive summands between $s\sigma$ and $t\sigma$, when $ndec > 1$. Assume $ndec > 1$ and let y and z be variables defined in the base segment:

1. If s is of the form $h(y+s')$ and t is of the form $h(z+s')$, it has to be guessed whether one adds either the equations (i) $y \geq z+ndec$ and $z \geq 0$, or the equations (ii) $y \leq z-ndec$ and $z < 0$, or the equations (iii) $y < 0$ and $z \geq 0$.
2. If s is of the form $h(y+s')$ and t is of the form $h(s')$, there is another choice between the equation (i) $y \geq ndec$, and the equation (ii) $y < 0$.
3. If s is of the form $h(x_{i+1}+y)$ and t is of the form $f(t')$, either the equation (i) $y \geq ndec - 1$ or (ii) $y < 0$ is added.

The following equations are added to the system D_S in order to express for which inequation which decisive summand is decisive, and whether it decides positively or negatively:

1. For each j between 1 and i, we guess which index summand k between 1 and $ndec$ is the decisive one for the inequation $x_j > x_{j+1}$. Now, for all $k' > k$ we add the equation $x_{j,k'} = x_{j+1,k'}$. In order to decide in which manner the k-th summand is decisive, we guess adding either (i) $x_{j,k} > x_{j+1,k} > 0$ or (ii) $x_{j,k} < x_{j+1,k}$ and $x_{j,k} < 0$.
2. Let $t^k_{j,l}$ be the result of replacing in $t_{j,l}$ every variable $x_{j'}$ by $x_{j',k}$, the integer variable corresponding to the k-th decisive summand. Now in order to make sure that the number of occurrences of the k-th summand at each side of the equality coincides, add $x_{j,k} = t^k_{j,l}$, for all j in $\{1 \ldots i\}$, and all k in $\{1 \ldots ndec\}$, and all l in $\{1 \ldots k_j\}$. We proceed identically with the associated equations.
3. We add $x_{i+1,1} = 1$, and for all k in $\{2 \ldots ndec\}$ we add $x_{i+1,k} = 0$.

Theorem 23 *The satisfiability problem for AG-RPO constraints restricted to signatures with functions of arity 0 or 1 is in NP.*

5.2 Arbitrary arities

The extension to arbitrary signatures is obtained analogously to the AC case. What has to be

taken into account is that $succsum_1(f(s_1,\ldots,s_k))$ is $h(0,\ldots,0,f(s_1,\ldots,s_k))$, and $succsum_1(h(s_1,\ldots,s_k))$ is $h(s_1,\ldots,s_k+sum_1)$ if s_k is not of the form $s'-sum_1$, and $h(s_1,\ldots,s_k-sum_1)$ otherwise.

Theorem 24 *The satisfiability problem for AG-RPO constraints is in NP.*

6 Hardness

Obviously, the satisfiability problems we deal with are NP-hard, because as subcases they include the AC, AC0 and AG-unifiability problems. But one may wonder whether there exists any ordering at all for these E such that at least the satisfiability problem for positive conjunctions of inequations (by which one cannot always encode unification) is in P. Here we answer this question negatively (by reducing 1-in-3-sat with only posive literals), even if monotonicity of the ordering is not required.

Theorem 25 *Let E be AC, AC0, or AG, and let $\succ$ be any arbitrary well-founded E-compatible ordering on ground terms that is total up to $=_E$. Then the constraint satisfiability problem for $\succ$ and $=_E$ is NP-hard even for constraints that are conjunctions of positive inequations.*

7 Conclusions and further work

Constraint solving algorithms have been defined for FRPO-based orderings for abelian semigroups, abelian monoids and abelian groups. We believe that the new techniques will lead to reasonably efficient practical algorithms for these orderings. This, as well as building an implementation, is one of the directions for further research in the context of the PhD. Thesis of the first author.

Finally, we expect that the ideas given here will provide new insights (to us or to others) for the development of constraint solving methods over fixed signatures for other E-compatible orderings.

References

[1] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1:329–349, 1985.

[2] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[3] H. Comon, R. Nieuwenhuis, and A. Rubio. Orderings, AC-Theories and Symbolic Constraint Solving. In *10th IEEE Symposium on Logic in Computer Science (LICS)*, pages 375–385, San Diego, USA, June 26–29, 1995.

[4] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 244–320. Elsevier Science Publishers B.V., Amsterdam, New York, Oxford, Tokyo, 1990.

[6] G. Godoy and R. Nieuwenhuis. Paramodulation with built-in abelian groups. In *15th IEEE Symposium on Logic in Computer Science (LICS)*, pages 413–424, Santa Barbara, USA, June 26–29, 2000. IEEE Computer Society Press.

[7] J.-P. Jouannaud and M. Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *18th International Colloquium Automata, Languages and Programming (ICALP)*, LNCS 510, pages 455–468, Madrid, Spain, July 16–20 1991. Springer-Verlag.

[8] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990.

[9] K. Korovin and A. Voronkov. A decision procedure for the existential theory of term algebras with the knuth-bendix ordering. In *15th IEEE Symposium on Logic in Computer Science (LICS)*, pages 291–302, Santa Barbara, USA, June 26–29, 2000. IEEE Computer Society Press.

[10] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, Dec. 1997.

[11] P. Narendran, M. Rusinowitch, and R. Verma. RPO constraint solving is in NP. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *12th Int. Conference of the European Association of Computer Science Logic (CSL'98)*, LNCS 1584, pages 385–398, Brno, Czech Republic, Aug. 23–28, 1999. Springer-Verlag.

[12] R. Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47:65–69, Aug. 1993.

[13] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering and Equality Constrained Clauses. *Journal of Symbolic Computation*, 19(4):321–351, April 1995.

[14] R. Nieuwenhuis and A. Rubio. Paramodulation with Built-in AC-Theories and Symbolic Constraints. *Journal of Symbolic Computation*, 23(1):1–21, May 1997.

[15] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers (to appear), 2000.

[16] A. Rubio and R. Nieuwenhuis. A total AC-compatible ordering based on RPO. *Theoretical Computer Science*, 142(2):209–227, May 15, 1995.

[17] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1987.

[18] L. Vigneron. Associative Commutative Deduction with constraints. In A. Bundy, editor, *12th International Conference on Automated Deduction (CADE)*, LNAI 814, pages 530–544, Nancy, France, June 1994. Springer-Verlag.

Invited Talk

Successive Approximation of Abstract Transition Relations *

Satyaki Das and David L. Dill
Computer Systems Laboratory,
Stanford University, CA 94305
satyaki@theforce.stanford.edu, dill@cs.stanford.edu

Abstract

Recently we have improved the efficiency of the predicate abstraction scheme presented in [7]. As a result the number of validity checks needed to prove the necessary verification condition has been reduced. The key idea is to refine an approximate abstract transition relation based on the counter-example generated. The system starts with an approximate abstract transition relation on which the verification condition (in our case this is a safety property) is model checked. If the property holds then the proof is done. Otherwise the model checker returns an abstract counter-example trace. This trace is used to refine the abstract transition relation if possible and start anew. At the end of the process the system either proves the verification condition or comes up with an abstract counter-example trace which holds in the most accurate abstract transition relation possible (with the user provided predicates as a basis). If the verification condition fails in the abstract system then either the concrete system does not satisfy it or the abstraction predicates chosen are not strong enough. This algorithm has been used on a concurrent garbage collection algorithm and a secure contract signing protocol. This method improved the performance on the first problem significantly and allowed us to tackle the second problem which the previous method could not handle.

1 Introduction

Abstraction is emerging as the key to formal verification of large designs, especially those that are not finite-state. *Predicate Abstraction* provides the potential for combining the generality of theorem proving with the ease-of-use of model checking by automatically mapping an unbounded system (called the *concrete system*) to a finite state system (called the *abstract system*). The states of the abstract system correspond to truth assignments to a set of *abstraction predicates*, which can be supplied by the user or derived from the problem using heuristics [4].

The user must supply a *verification condition* that is to be proved. Throughout this paper, the verification condition is assumed to be an invariant. Of course more complex safety properties can be checked by augmenting the system description with history variables, and specifying an invariant over the history variables. Either the system extracts appropriate predicates or uses user provided abstraction predicates to automatically construct an abstract system from the concrete system description.

Model checking techniques can then be used to check whether the abstract system satisfies the verification condition. The abstraction is *conservative*, meaning that if a property is shown to hold on the abstract system, there is a concrete version of the property that holds on the concrete system; however, if the verification condition fails to hold on the abstract system, it may or may not hold on the concrete system.

The prototype system described here handles more complex system descriptions than methods previously described. It uses two existing libraries: SVC [2], an implementation of decision procedures for quantifier-free first-order logic, and Boolean Decision Diagrams (called BDDs), an efficient representation for Boolean functions. The use of these efficient libraries is crucial for the success of the system. For example, SVC is typically called tens of thousands of times during verification.

The prototype works in two phases: it first produces a representation of a finite-state machine that is a conservative abstraction of the concrete system. Creating a good abstract machine is expensive, so an over-approximation of the abstract transition relation is computed. In the second phase, the verification condition is checked on this machine using a variant of standard BDD-based model checking algorithms. If the verification condition holds then the proof is complete. Otherwise an abstract counter-example trace is generated. This counter-example is checked to see

*This work was supported by NASA contract NAS1-98139 and DARPA contract 00-C-8015. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

0-7695-1281-X/01 $10.00 © 2001 IEEE

whether it is an artifact of the approximation during the first phase. If it is, then the abstract transition relation is refined (by adding constraints to the transition relation) so as to eliminate the spurious counter-example and the verification condition is model checked once again. This process is repeated until the verification condition is proved or a valid abstract counter-example is generated. This *counter-example guided refinement* phase is essential to speed up the predicate abstraction process.

The technique has been applied to a concurrent garbage collection algorithm and a contract signing protocol. The new technique was able to verify the garbage collection algorithm much faster than the technique used by Das, Dill, and Park in 1999 [7], which was the first and still only attempt to verify it using predicate abstraction. The original method could not even prove the contract signing protocol because the proof obligations generated were too difficult for the decision procedure.

Related work

The use of automatic predicate abstraction for model checking infinite-state systems was first presented by Graf and Saïdi in 1997 [9]. Their method represented the abstract states as *monomials* (monomials are conjunctions of abstract state variables or their negations). Compared with the original method of Das, Dill, and Park, and the new method, the use of monomials may result in more false errors and failed proofs. However their method also requires fewer validity checks. The original Graf/Saïdi method computes the reachable state set as part of the abstraction process. Our work uses some of the ideas present in the Graf/Saïdi abstraction scheme [9] and [7].

The creation of the initial abstract transition relation is similar to the abstraction method presented by Saïdi and Shankar [15]. In that work the authors construct an accurate abstract transition relation that is used in model checking. If the desired invariant does not hold, then new predicates are added. In their paper, refinement is used to construct the new abstract transition relation from the original relation. Their method computes the exact abstract transition relation which can be expensive. In contrast our strategy of successive approximation is more efficient because it attempts to compute the *least* accurate approximation that gives a definitive answer.

Colón and Uribe have also described a method that first generates an abstract transition system, then model checks it [6]. The transition relation generated is less accurate than that presented here.

The idea of counter-example-guided refinement is a generally useful technique in model checking, which has been used before, by Kurshan *et al.* [13] for checking timed automata, Balarin *et al.* [1] for language containment and Clarke *et al* [5] in the context of verification using abstraction for different variables in a version of the SMV model checker. Counter-example guided refinement has even been used with predicate abstraction by Lakhnech *et al.* [18]. However, their method refines by discovering new predicates to add, an idea that is quite different from refining the *use* of a given set of predicates in the abstract system.

We believe that the present method can handle significantly larger problems than previous methods. So far as we know, the original method of Das, Dill and Park is able to handle more difficult problems than any of the other methods described above, and the new method is much more efficient.

2 Abstraction Method

This section summarizes the theory of *conservative* abstraction. Since the theory behind this is well known and descriptions of this can be found in previous papers on this subject (for instance in [9]), the important properties of the abstraction will mostly be stated without formal proof. In stating and proving the claims, we have found that using logical formulas uniformly, instead of a mix of set and logic notation, eliminates a certain amount of confusion. Hence initial states, transition relations and reachable state sets are represented as predicates.

The key idea in conservative abstraction is that the abstract state machine yields a superset of the reachable concrete states. This means that if the verification condition holds in the superset of the reachable concrete states then it will also hold in the concrete system.

The concrete transition system consists of initial states represented by the predicate I_C. $I_C(x)$ is $true$ iff x is an initial state. The transition system is represented by $R_C(x, y)$. $R_C(x, y)$ is $true$ iff y is a successor of x.

The concrete system is mapped to an abstract transition system. If there are N abstraction predicates, $\phi_1, \phi_2, \ldots \phi_N$, then the abstract state space is the subset of all bit-vectors of length N, which can be modeled as follows. If $P = \{x \in N \mid 0 < x \leq N\}$, then the type of these bit-vectors is $P \rightarrow \{0, 1\}$. In what follows 1 and 0 shall be interpreted as $true$ and *false* in the obvious way. The initial states and the transition relation for the abstract system are constructed later in the section.

The abstraction can be formalized as a standard Galois connection, having an abstraction function, α which maps concrete states to bit-vectors, and a concretization function, γ which is essentially the inverse image of α. Specifically, $\alpha(x)$ is a bit-vector whose i^{th} bit has the truth value $\phi_i(x)$ while $\gamma(s)$ is a predicate on concrete states that hold on x when for every $i \in P$ the i^{th} bit of s matches $\phi_i(x)$.

Definition 1 *The abstraction and concretization functions,*

$\alpha : C \to (P \to \{0,1\})$ *and* $\gamma : (P \to \{0,1\}) \to C$ *are defined as,*

$$\alpha(x)(i) = \phi_i(x)$$
$$\gamma(s)(x) = \bigwedge_{i \in P} \phi_i(x) \equiv s(i)$$

($\equiv$ is the biconditional)

The definition of α and γ can be extended to work on the predicates defined over the concrete states and abstract states respectively. These extended definitions are as follows:

Definition 2 *Given predicates, Q_C and Q_A over concrete and abstract states respectively, the abstraction and concretization functions are extended as follows:*

$$\alpha(Q_C)(s) = \exists x.\, Q_C(x) \wedge \bigwedge_{i \in P} \phi_i(x) \equiv s(i)$$
$$\gamma(Q_A)(x) = \exists s.\, Q_A(s) \wedge \bigwedge_{i \in P} \phi_i(x) \equiv s(i)$$

Predicates are used to describe sets. So the set of all abstract states are defined by the predicate, $\exists x.\, \gamma(s)(x)$. Then for any arbitrary predicates S and X defined on the abstract and concrete states respectively it can be easily proved that,

$$X \to \gamma(\alpha(X))$$
$$(\exists x.\, \gamma(S)(x)) \to (S = \alpha(\gamma(S)))$$

These two results show that the abstraction scheme is indeed a Galois connection.

Definition 3 *The set of abstract initial states, I_A is defined to be $\alpha(I_C)$.*

Notice that α has been used on a concrete predicate and so the second definition of α is to be used. It may be shown that the concrete and abstract initial states satisfy the inclusion relation, $I_C \to \gamma(I_A)$

Definition 4 *The abstract transition relation is represented by a predicate R_A with two states, s and t as arguments. The transition relation is defined as,*

$$R_A(s,t) = \exists x, y.\ \gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x,y)$$

The abstract transition system so defined is a conservative abstraction of the concrete system. Let the predicate $S_A^k(s)$ hold if s is an abstract state that is reachable from an initial state after k transitions. Similarly let the predicate $S_C^k(x)$ hold if x is a concrete state that is reachable from an initial state after k transitions. Assuming that

$$\forall x.\ S_C^k(x) \to \gamma(S_A^k)(x) \qquad (1)$$

holds it can easily be shown that

$$\forall x.\ S_C^{k+1}(x) \to \gamma(S_A^{k+1})(x)$$

where the reachable concrete and abstract states after $k+1$ transitions are given by

$$S_C^{k+1}(y) = S_C^k(y) \vee \exists x.\, S_C^k(x) \wedge R_C(x,y)$$
$$S_A^{k+1}(t) = S_A^k(t) \vee \exists s.\, S_A^k(s) \wedge R_A(s,t)$$

Then by induction it may be concluded that (1) holds for all k. Since the abstract system is finite, the fixed point of abstract reachable states exists and the concretization of the abstract reachable states must include all concrete reachable states. This shows that any invariant that holds in the concretization of the abstract reachable states must also hold in the concrete system. Thus the abstract system is a conservative abstraction of the concrete system.

3 Counter-Example Guided Refinement

Now that the abstract system has been defined, a method is presented to compute the abstract system efficiently and with the necessary accuracy. Usually, computing the exact abstract transition relation defined in the previous section requires excessive time for all but the most trivial of systems. Also typically the set of abstract reachable states is extremely sparse. So most of the abstract states are unreachable. Hence computing the full transition relation is not necessary.

Assume that the successive approximation process starts with an over-approximation, R_0, of the exact abstract transition relation. If a state t is a successor of s in the exact transition relation then t is also a successor of s in the over approximated transition relation as well. R_0 is used to model check the verification condition. If the verification condition holds then the proof is complete. Otherwise the model checker generates an *abstract counter-example trace* which violates the verification condition. The abstract counter-example trace is a finite sequence of abstract states, $s_0, s_1, \ldots s_n$ such that $I_A(s_0)$ holds and $R_0(s_i, s_{i+1})$ holds for every $i \in [0,n)$. Also s_n violates the verification condition. Now, for each pair of consecutive abstract states, (s_i, s_{i+1}), check if $R_A(s_i, s_{i+1})$ holds. In this case, a valid abstract counter-example has been found. Otherwise R_0, can be refined to eliminate the generated counter-example. This process of model checking followed by refinement is repeated till the verification condition is proved or a valid counter-example is found.

We now explain how the refinement process works. Suppose R is the an over approximated abstract transition relation and that the abstract counter-example trace found after model checking has two consecutive states, s_j and s_{j+1}, such that $R_A(s_j, s_{j+1})$ is *false*. The algorithm tries to find

```
PROVE_VERIFICATION_CONDITION(property)
begin
  I_A := Initial State predicate
  R_A := true
  while (true)
    R_orig := R_A
    trace := model check property in abstract system, (I_A, R_A)
    if empty(trace) then
      return PROPERTY_PROVED
    else
      for each pair of successive states s_j, s_{j+1} in trace do
        if γ(s_j)(x) ∧ γ(s_{j+1})(y) ∧ R_C(x,y) is unsatisfiable
        then
          R_orig := R_A
          R_A := R_A ∧ REFINE_TRANS_REL(s_j, s_{j+1})
          break
        endif
      end
      if R_A = R_orig return trace
    endif
  end
end

REFINE_TRANS_REL(s_j, s_{j+1})
/* The function returns the constraint C */
begin
  X := γ(s_j)(x) ∧ γ(s_{j+1})(y)
  for each conjunct, p in X do
    remove p from X
    if satisfiable(X ∧ R_C(x,y)) then
      add p back to X
    endif
  end
  return ¬α(X)
end
```

Figure 1. Abstract State Machine Refinement

a constraint, $C(s,t)$, such that $R_A(s,t) \rightarrow C(s,t)$ and $C(s_j, s_{j+1})$ is *false*. Then the abstract transition relation,

$$R'(s,t) = R(s,t) \wedge C(s,t)$$

is also a conservative abstract transition relation. Since $R_A(s_j, s_{j+1})$ is *false*, this means that $\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$ is unsatisfiable for every x and every y. From the definition of γ, it follows that $\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y)$ is a conjunction of abstraction predicates, $\phi_i(x)$ and $\phi_i(y)$ and their logical complements. We wish to find a minimal subset of these predicates that is unsatisfiable when conjoined with $R_C(x,y)$. The heuristic in the present system is a simple greedy algorithm. It is explained in Figure 1.

The following theorem shows that this construction results in a new conservative abstract transition relation. The key point to note is that at the end of the algorithm the conjunction of the remaining conjuncts and $R_C(x,y)$ is unsatisfiable. The bit-vectors c_j and c_{j+1} determine which conjuncts have been removed. Wherever $c_j(k)$ is *false*, the conjunct involving $\phi_k(x)$ has been removed from $\gamma(s_j)(x)$ in the added constraint, $C(s,t)$. Similarly, if $c_{j+1}(k)$ is false, then the conjunct involving $\phi_k(y)$ has been removed from $\gamma(s_{j+1})(y)$.

Theorem 1 *Let the initial abstract transition relation, R satisfy $\forall s,t.\ R_A(s,t) \rightarrow R(s,t)$ and s_j, s_{j+1} be abstract states and c_j and c_{j+1} are bit-vectors such that*

$$\bigwedge_{i \in P} c_j(i) \rightarrow (s_j(i) \equiv \phi_i(x))$$
$$\wedge \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (s_{j+1}(i) \equiv \phi_i(y)) \wedge R_C(x,y)$$

is unsatisfiable, then the new transition relation defined by,

$$\begin{aligned} R'(s,t) \;=\; & R(s,t) \wedge \\ & \neg[\bigwedge_{i \in P} c_j(i) \rightarrow (s(i) \equiv s_j(i)) \wedge \\ & \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (t(i) \equiv s_{j+1}(i))] \end{aligned}$$

satisfies

$$\forall s,t.\ R_A(s,t) \rightarrow R'(s,t)$$

Proof To prove the theorem assume that $R_A(s,t)$ holds for some arbitrary s and t.

Since $R_A(s,t) \rightarrow R(s,t)$, it may be concluded that $R(s,t)$ holds as well. Also by definition of R_A,

$$\exists x,y.\ \gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x,y)$$

Existential instantiation of the quantifier and using the definition of γ yields,

$$\bigwedge_{i \in P} s(i) \equiv \phi_i(x_0) \wedge \bigwedge_{i \in P} t(i) \equiv \phi_i(y_0) \wedge R_C(x_0, y_0) \quad (2)$$

Because of the condition that c_j and c_{j+1} satisfies,

$$\begin{aligned} \neg[\exists x,y.\ & \bigwedge_{i \in P} c_j(i) \rightarrow (s_j(i) \equiv \phi_i(x)) \wedge \\ & \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (s_{j+1}(i) \equiv \phi_i(y)) \wedge R_C(x,y)] \end{aligned}$$

Simplifying the expression and then instantiating with x_0 and y_0 yields,

$$\begin{aligned} & [\bigvee_{i \in P} c_j(i) \wedge (s_j(i) \not\equiv \phi_i(x_0))] \\ \vee\ & [\bigvee_{i \in P} c_{j+1}(i) \wedge (s_{j+1}(i) \not\equiv \phi_i(y_0))] \\ \vee\ & \neg R_C(x_0, y_0) \end{aligned}$$

Using the expressions for $\phi_i(x_0)$ and $\phi_i(y_0)$ from (2) yields,

$$\bigvee_{i \in P} c_j(i) \wedge (s_j(i) \not\equiv s(i)) \\ \vee \bigvee_{i \in P} c_{j+1}(i) \wedge (s_{j+1}(i) \not\equiv t(i)) \quad (3)$$

Now from the definition of R',

$$R'(s,t) = R(s,t) \wedge \neg[\bigwedge_{i \in P} c_j(i) \rightarrow (s(i) \equiv s_j(i)) \wedge \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (t(i) \equiv s_{j+1}(i))]$$

Simplifying the above definition and using that $R(s,t)$ holds,

$$R'(s,t) = [\bigvee_{i \in P} c_j(i) \wedge (s(i) \not\equiv s_j(i)) \vee \bigvee_{i \in P} c_{j+1}(i) \wedge (t(i) \not\equiv s_{j+1}(i))] \quad (4)$$

The combination of (4) and (3) shows that $R'(s,t)$ holds. This completes the proof of the theorem. □

As mentioned above, the approximate abstract system is model checked, and then refined if necessary. This process is repeated until one of the following happens:

1. The verification condition holds.
2. A counter-example trace in which for any two successive states, s_j and s_{j+1},
   $\exists x, y.\ \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$
   holds.

It is easy to see that the process will necessarily terminate in one of these situations. Every refinement must remove at least one pair of abstract states from the transition relation. Since the abstract system is finite, the number of times the refinement can be carried out is bounded.

In the first scenario the desired invariant holds in an over-approximation of the exact abstract transition relation and so would also hold in the exact transition relation. Thus the desired invariant has been proved correct. In the second case the counter-example generated would also hold in the abstract machine with transition relation R_A. So further refinement of R_A would be useless. This is proved in the next theorem.

Theorem 2 *If an abstract transition system with transition relation, R such that $R_A \rightarrow R$ and initial state set, I_A has a counter-example trace, $s_0, s_1, \ldots s_n$ such that for each $j \in [0,n)$ there are concrete states x and y (not necessarily the same for different values of j) such that,*

$$\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$$

is satisfiable, then $s_0, s_1, \ldots s_n$ is also a counter-example trace in the abstract transition system where the transition relation is R_A and the initial state set is I_A.

Proof Since $s_0, s_1, \ldots s_n$ is an execution trace in the approximate transition system,

$$I_A(s_0) \quad (5)$$

Now for every $j \in [0,n)$,

$$R_A(s_j, s_{j+1}) = \exists x, y.\ \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y) \quad (6)$$

Existential instantiation of the precondition of the theorem yields,

$$\gamma(s_i)(x_0) \wedge \gamma(s_{i+1})(y_0) \wedge R_C(x_0, y_0)$$

Using this with (6) implies that $R_A(s_j, s_{j+1})$ is *true* and so s_{j+1} is a successor of s_j. Using this fact in conjunction with (5) proves that $s_0, s_1, \ldots s_n$ is a counter-example trace in the exact abstract system. □

Thus, if a counter-example is generated, either the set of predicates provided are not rich enough to prove the desired verification condition or the invariant does not hold in the concrete system.

4 Prototype Implementation and Results

A prototype verifier based on the preceding ideas was implemented to evaluate efficiency on real problems. The decision procedure, SVC was used to do the satisfiability checks. Binary Decision Diagrams were used to represent the abstract transition relation and to model check the verfication condition on the abstract system. The user has to provide the predicates used to construct the abstract system.

An obvious choice for the initial approximate abstract transition relation is the completely unconstrained abstract transition relation. The decision procedure, SVC, did not perform well when this was the case, so the prototype produced an initial approximation by heuristically collecting small sets of predicates with many common variables, and building a abstract transition relation using only those predicates.

Unlike the preceding discussion, the prototype creates abstraction predicates on the next-state variables by substituting transition functions for current state variables in the abstraction functions (this is the method used in most previous papers on predicate abstraction).

We have used two examples to evaluate the successive approximation method presented here. The examples are:

- On-The-Fly Garbage Collection
- GJM Secure Contract Signing Protocol

On-The-Fly Garbage Collection

The on-the-fly garbage collection algorithm was proposed by Dijkstra, et al. [8]. This algorithm is widely acknowledged to be difficult to get right, and difficult to prove. A more detailed discussion of the subtlety of this algorithm and subsequent variations can be found in a paper by Havelund and Shankar [10].

The algorithm was simplified by Ben-Ari [3] to involve two colors instead of three. This also led to a simpler argument of correctness. Alternative justifications of Ben-Ari's algorithm were also given by Van de Snepscheut [17] and Pixley [12]. However it must be remembered that these proofs were informal *pencil and paper* proofs.

Later this modified algorithm was mechanically proved by Russinoff [14] using the Boyer-Moore theorem prover. A formulation of the same algorithm was also proved by Havelund and Shankar in PVS [10]. The authors give an estimation of the complexity and size of the proof. The proof needed 19 invariant lemmas and 57 function lemmas and took about two months. So far as we know, no one has mechanically proved the original algorithm of Dijkstra, et al.

In the garbage collection algorithm, the *collector* and the user program, the *mutator*, may be regarded as a concurrent system with both processes working on shared memory. The memory is abstractly modeled as a directed graph with each node having at most two outgoing edges. A subset of these nodes are called *roots* and they are special in the sense that they are always accessible to the mutator. Also any node that can be reached from one of the roots by following edges is also accessible to the mutator. The mutator is allowed to choose an arbitrary node and redirect one of its edges towards another arbitrarily chosen accessible node. Each memory node also has a *color* field which the collector uses to keep track of the accessible nodes. The collector also maintains a *free-list* which is a list of nodes that are not being used by the mutator. The mutator can request nodes from the collector which the collector satisfies from the free-list. The collector collects *garbage* nodes (that is nodes which are no longer accessible to the mutator) and adds them to the free-list.

The garbage collection algorithm must satisfy two properties for it to be correct. First it must guarantee that no node accessible to the mutator is ever added to the free-list. The second property is that if some node becomes inaccessible to the mutator it is eventually added to the free-list. The first property makes sure that no data which would be used by the user program is ever freed. The second property makes sure that there are no memory leaks in the system. We have proved that the first property holds for the algorithm using predicate abstraction. The proof of correctness needs some auxillary graph properties which are treated as axioms by the predicate abstraction tool.

GJM Abuse-Free Contract Signing Protocol

The abuse-free contract signing protocol provides a mechanism for signing contracts between two parties and guarantees some correctness properties. A *contract* can be thought of as reciprocal promises between the involved parties. For instance if Alice is buying a car from Bob then she promises to pay Bob the negotiated price while he promises to give her the car.

A very basic correctness condition is *fairness*. For a contract signing protocol to be fair it must be the case that after the protocol terminates either both parties have a contract or neither party has a contract. In the previous example if Alice promises to pay the price of the car she should have a promise from Bob that he would give her the car. Otherwise the protocol violates fairness.

Other correctness properties of the protocol are *accountability* and *abuse-freeness*. We have not proved these properties.

The protocol we have studied here was introduced in [11]. The protocol depends on a *trusted third party* to resolve conflicts. The protocol works in two phases. In the first phase the participants exchange messages and try to arrive at a contract. If something goes wrong (either because messages were lost or because of foul play) the trusted third party resolves the contract. The protocol has been exhaustively analyzed for weaknesses using a model checker [16] with a finite number of concurrent contract signings. A problem was discovered during this and was fixed. We have looked at the fixed protocol and proved that it maintains fairness with any number of concurrent contract signings.

Results

For each example, the execution times on a 800MHz Pentium processor are reported. In the table below the abstraction time is the time required to compute the initial approximate transition relation. The model checking time is the time required to repeatedly model check and refine the abstraction. The time required is compared to the approach presented in implicit predicate abstraction [7].

One reason that the current method works much better than implicit predicate abstraction is that it never has to check the satisfiability of similar expressions repeatedly. To see why this can be a problem with implicit predicate abstraction consider the following example. Assume that we have abstraction predicates $\phi_1 \equiv a > b$ and $\phi_2 \equiv b > a$ (where a and b are concrete state variables). It is obvious that both predicates can not be true at the same

	Abstraction time (in hr:min)	Model checking time (in min)
GC(implicit)	2:25	N/A
GC(current)	0:09	1
GJM(implicit)	24hr+	N/A
GJM(current)	0:13	4

time. In the implicit abstraction scheme, expressions, which are unsatisfiable because they are conjunctions containing $\phi_1(x) \wedge \phi_2(x)$, are checked for satisfiability repeatedly. In the current method this will be recognized the first time a counter-example has both predicates true. After that the abstract transition relation will be suitably modified so that a counter-example is never generated which has both predicates asserted simultaneosly.

Another interesting observation is that the set of reachable abstract states is usually extremely sparse. So the current method will perform much better than systems which naively compute an exact abstract transition system.

If the verification condition can be proved with the provided abstraction predicates then the current method will indeed be able to prove the verification condition. Thus if the proof fails then that means that the set of abstraction predicates is not enough to prove the verification condition. In systems which construct a weaker abstraction, a failed proof has to be investigated to determine if the proof failed because the abstraction predicates are insufficient or because the approximation lost information.

5 Conclusion

This paper demonstrates that using counter-example guided refinement with predicate abstraction can reduce the computational difficulty of formally verifying systems with unbounded numbers of states. However, we have only done a few examples of any size, and there are obviously many additional problems that would need to be solved before predicate abstraction could be used as routinely as model checking is currently.

The most obvious issue at this point is the need to find good candidate predicates automatically, instead of requiring the user to provide them. This problem has been addressed to some extent by others (as discussed in section 1), but it is not clear that the techniques would scale up to the size of problems in the previous section. Automatically deriving excessively complex predicates or too many irrelevant predicates could make the computational part of predicate abstraction too difficult. Another important issue is how to find good candidate predicates containing quantifiers, which are needed for the examples in the previous section.

Another difficult issue is how to discover when there are design errors. A good pragmatic step would be to model check a finite instance of the problem before applying predicate abstraction. But feasible finite instances may not exhibit the errors (which is the motivation for doing predicate abstraction in the first place). In the system described here, errors will result in valid abstract counter-examples, but there is no algorithmic way to determine if these correspond to a concrete counter-example, which is what the user really needs to determine whether the problem is a design error or an inadequate abstraction. Of course, the problem is undecidable, so there is no perfect solution, but there may be good heuristics for finding useful counter-examples.

References

[1] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *5th International Conference on Computer-Aided Verification*, pages 29–40, 1993.

[2] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

[3] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[4] S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A tool for the verification of invariants. In *10th International Conference on Computer-Aided Verification*, pages 505–510, 1998.

[5] E. *et al.*. Clarke. Counterexample-guided abstraction refinement. In *12th International Conference on Computer-Aided Verification*, pages 154–169, July 2000.

[6] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1998.

[7] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[8] E. W. Dijkstra, L. Lamport, A. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–75, November 1978.

[9] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Conference on Computer Aided Verification*, volume 1254 of *Lecture notes in Computer Science*, pages 72–83. Springer-Verlag, 1997. June 1997, Haifa, Israel.

[10] K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. Available at http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund, 1997.

[11] M. J. J. A. Garay and P. MacKenzie. Abuse-free optimistic contract signing. In *Proc. Advances in Cryptology - Crypto '99*, pages 449–466, 1999.

[12] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3(1):41–50, 1988.

[13] R. K. R. Alur, A. Itai and M. Yannakakis. Timing verification by successive approximation. *Information and Computation 118(1)*, pages 142–157, 1995.

[14] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6(4):359–390, 1994.

[15] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[16] V. Shmatikov and J. C. Mitchell. Analysis of abuse-free contract signing. In *Financial Cryptography*, 2000. Anguilla.

[17] J. van de Snepscheut. Algorithms for on-the-fly garbage collection revisited. *Information Processing Letters*, 24(4):211–16, March 1987.

[18] S. B. Y. Lakhnech, S. Bensalem and S. Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

Session 2

A Bound on Attacks on Payment Protocols

Scott D. Stoller*
Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400 USA

Abstract

Electronic payment protocols are designed to work correctly in the presence of an adversary that can prompt honest principals to engage in an unbounded number of concurrent instances of the protocol. This paper establishes an upper bound on the number of protocol instances needed to attack a large class of protocols, which contains versions of some well-known electronic payment protocols, including SET and 1KP. Such bounds clarify the nature of attacks on and provide a rigorous basis for automated verification of payment protocols.

1. Introduction

Many protocols, including electronic payment protocols, are designed to work correctly in the presence of an adversary (also called a penetrator) that can prompt honest principals to engage in an unbounded number of concurrent instances of the protocol. Payment protocols should satisfy at least two kinds of correctness requirements: *secrecy*, which states that certain values are not obtained by the penetrator, and *agreement*, which states that a principal executes a certain action only if appropriate other principals previously executed corresponding other actions (*e.g.*, a payment gateway approves a charge to customer C's account only if customer C previously authorized that charge).

Allowing an unbounded number of concurrent protocol instances makes the number of reachable states unbounded. The case studies in, *e.g.*, [13, 6, 19, 10, 17] show that state-space exploration of security protocols is feasible when small upper bounds are imposed on the size of messages and the number of protocol instances. In most of those case studies, the bounds are not rigorously justified, so the results do not prove correctness of the protocols. Rigorous automated verification of these protocols requires either symbolic state-space exploration algorithms that directly accommodate these infinite state spaces or theorems that reduce correctness of these protocols to finite-state problems.

This paper presents a reduction for a large class of protocols. It uses the strand space model [24]. A regular strand can be regarded as a thread that runs the program corresponding to one role of the protocol and then terminates. A central hypothesis of our reduction is the bounded support restriction (BSR), which states that in every history (*i.e.*, every possible behavior) of the protocol, each regular strand depends on at most a given number of other regular strands. Our notion of dependence, embodied in the definition of *support*, is a variant of Lamport's happened-before relation [15], modified to handle freshness of nonces appropriately. BSR is not easily checked by static analysis, so we propose to check it by state-space exploration, while checking the correctness requirements. With statically checkable restrictions alone, it seems difficult to find restrictions that are both strong enough to justify a reduction and weak enough to be satisfied by well-known protocols.

To check BSR by state-space exploration, we need a reduction for it. We prove: if a protocol satisfies its correctness requirements and BSR when appropriate bounds are imposed on the number of regular strands in a history, then the protocol also satisfies its correctness requirements and BSR without those bounds.

Most existing techniques for automated analysis of systems with unbounded numbers of concurrent processes, such as [9, 11, 2, 3, 14], are not applicable to payment protocols, because they assume the set of values (equivalently, the set of local states of each process) is independent of the number of processes, whereas payment protocols generate fresh values, so the set of values grows as the number of processes (strands) increases.

Roscoe and Broadfoot use data-independence techniques to bound the number of nonces needed for an attack [20]. Their result assumes that each trustworthy principal participates in at most a given number of protocol instances at a time. Our reduction does not require that assumption; indeed, our goal is to justify such assumptions. Lowe's reduction [16] has similar goals as our reduction and provides tighter bounds in its domain of applicability, but it does not handle agreement requirements and does not apply to the variants of SET and 1KP described in Section 2.1.

The reduction embodied in Theorems 2 and 3 handles secrecy and agreement requirements and applies to simplified versions of SET [21] and 1KP [4]. It extends the reduction in [22] in several significant ways. The class of preserved properties is extended to allow protocol-specific secrecy properties (roughly, any non-cryptographic value can

*The author gratefully acknowledges the support of NSF under Grant CCR-9876058 and the support of ONR under Grants N00014-99-1-0358 and N00014-01-1-0109. Email: stoller@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~stoller/ Phone: 631-632-1627

0-7695-1281-X/01 $10.00 © 2001 IEEE

be designated as a secret) and to allow use of more general predicates to characterize the desired relationship between actions in agreement properties. The class of protocols is extended by allowing hash functions, allowing arbitrary nesting of hashing and encryption in protocol messages, and relaxing the restriction that the recipient of a message be able to recognize the entire structure of the message.[1] These extensions necessitate substantial changes to the statement and proof of Theorem 1. That theorem is the crux of the proof of our reduction: it provides a statically-calculated bound on a "dynamic" quantity (*i.e.*, a quantity defined by a maximum over all possible executions of the protocol); that quantity is the dependence width, defined in Section 4.

Our results implicitly describe a simulation relation between systems with bounded-size histories and systems with unbounded-size histories. It would be interesting to see whether similar results could be obtained more easily in a process-algebraic framework, such as Spi calculus [1].

2. Model of Protocols

We use the strand space model [24], with minor modifications.

The set of *primitive terms* is $Prim = Text \cup Key$, where $Text$ is a set of values other than cryptographic keys, and $Key = \{key(x,y) \mid x,y \in Name \wedge x \neq y\} \cup \{pub(x) \mid x \in Name\} \cup \{pvt(x) \mid x \in Name\}$. Informally, $key(x,y)$ is a symmetric key intended for use by x and y, and $pub(x)$ and $pvt(x)$ represent x's public and private keys, respectively, in a public-key cryptosystem. $Name$ is the subset of $Text$ containing names of principals. $Nonce$ is the subset of $Text$ containing nonces.

The set $Term$ of terms is defined inductively as follows. (1) All primitive terms are terms. (2) If t and t' are terms and $k \in Key$, then $encr(t,k)$ (encryption of t with k, usually written $\{t\}_k$), $pair(t,t')$ (pairing of t and t', usually written $t \cdot t'$), and $h(t)$ (hash of t, where h represents a one-way collision-resistant hash function [18]) are terms.

$\mathrm{inv} \in Key \to Key$ maps each key to its inverse: decrypting $\{t\}_k$ with $\mathrm{inv}(k)$ yields t. For a symmetric key k, $\mathrm{inv}(k) = k$. We usually write $\mathrm{inv}(k)$ as k^{-1}.

$[t]_{pvt(x)}$ abbreviates $t \cdot \{h(t)\}_{pvt(x)}$, *i.e.*, t signed by x.

A *ciphertext* is a term whose outermost operator is $encr$. A *hash* is a term whose outermost operator is h. A term t' *occurs in the clear* in t if there is an occurrence of t' in t that is not in the scope of $encr$ or h.

Let $\mathrm{dom}(f)$ denote the domain of a function f. A sequence is a function whose domain is a finite prefix of the natural numbers. Let $\mathrm{len}(\sigma)$ denote the length of a sequence σ. $\langle\!\langle a, b, \ldots \rangle\!\rangle$ denotes a sequence σ with $\sigma(0) = a$, $\sigma(1) = b$, and so on.

A *directed term* is $+t$ or $-t$, where t is a term. Positive and negative terms represent sending and receiving messages, respectively. We sometimes refer to directed terms as "terms" and treat them as terms, for instance as having subterms.

A *trace* is a finite sequence of directed terms. Let $Trace$ denote the set of traces.

A *trace mapping* is a function $tr \in \mathrm{dom}(tr) \to Trace$, where $\mathrm{dom}(tr)$ is an arbitrary set whose elements are called *strands*.

A *node* of tr is a pair $\langle s, i \rangle$ with $s \in \mathrm{dom}(tr)$ and $0 \leq i < \mathrm{len}(tr(s))$. Let $\mathcal{N}_{tr}$ denote the set of nodes of tr. We say that node $\langle s, i \rangle$ is on strand s. Let $\mathrm{nodes}_{tr}(s)$ denote the set of nodes on strand s in tr. Let $\mathrm{strand}(\langle s, i \rangle) = s$, $\mathrm{index}(\langle s, i \rangle) = i$, and $\mathrm{term}_{tr}(\langle s, i \rangle) = tr(s)(i)$.

The local dependence relation is: $\langle s_1, i_1 \rangle \stackrel{lcl}{\to} \langle s_2, i_2 \rangle$ iff $s_1 = s_2$ and $i_2 = i_1 + 1$.

A term t *originates* from a node $\langle s, i \rangle$ in tr iff $\langle s, i \rangle$ is positive, t is a subterm of $\mathrm{term}_{tr}(\langle s, i \rangle)$, and t is not a subterm of $\mathrm{term}_{tr}(\langle s, j \rangle)$ for any $j < i$.

A term t *uniquely originates* from a node n in tr iff it originates from n in tr and not from any other node in tr. Typically, nonces are uniquely-originated. This is the strand space way of expressing freshness.

For $S \subseteq \mathcal{N}_{tr}$, let $\mathrm{term}_{tr}(S) = \{\mathrm{term}_{tr}(n) \mid n \in S\}$. For symbols subscripted by the trace mapping, we elide the subscript when the trace mapping is evident from context.

2.1. Roles, Protocols, and Penetrator

A *role* is a parameterized sequence of directed terms. Associated with each parameter is a type, *i.e.*, a set of allowed terms. Some parameters with type $Nonce$ may be designated as uniquely-originated; informally, this means that the value of that parameter must be uniquely-originated. Uniquely-originated parameters are designated by underlining in the parameter list. We require that for every role r, for every parameter x of r with type $Nonce$, x is uniquely-originated iff the first occurrence of x in r is in a positive term. Let $r.x$ denote parameter x of role r. For example, $R(\underline{nc} : Nonce) = \langle\!\langle +nc \rangle\!\rangle$ defines a role R with one uniquely-originated parameter nc with type $Nonce$.

A *trace for role* r is a prefix of a trace obtained by substituting for each parameter x of r a term in the type of x. A role r and a trace σ for r uniquely determine a mapping, denoted $args(r, \sigma)$, from the set of parameters of r that appear in $r(0), r(1), \ldots, r(\mathrm{len}(\sigma) - 1)$ to $Term$. For example, for role $R(x_1 : Name, x_2 : Name) = \langle\!\langle +x_1, +x_2 \rangle\!\rangle$ and $\sigma = \langle\!\langle +A \rangle\!\rangle$, $\mathrm{dom}(args(R, \sigma)) = \{x_1\}$ and $args(R, \sigma)(x_1) = A$.

[1] Session keys are not used in the examples in this paper, so we omitted them from the framework. They can be handled roughly as in [22].

A *protocol* Π is a set of roles, together with a set $\Pi.Secret \subseteq (Text \setminus (Name \cup Nonce))$ of terms that are "secrets" (*i.e.*, terms that should not be revealed to the penetrator). Excluding names here implies that the penetrator knows all names. Specialized notions of secrecy are used for keys and nonces, as described in Section 2.5.

The penetrator model is parameterized by a set $Key_P \subset Key$ of keys initially known to the penetrator. The set $\Pi_P(Key_P)$ of *penetrator roles* contains:

Pair: $P(x : Term, y : Term) = \langle\!\langle -x, -y, +x{\cdot}y \rangle\!\rangle$
Separation: $S(x : Term, y : Term) = \langle\!\langle -x{\cdot}y, +x, +y \rangle\!\rangle$
Encryption: $E(k : Key, x : Term) = \langle\!\langle -k, -x, +\{x\}_k \rangle\!\rangle$
Decryption: $D(k{:}Key, x{:}Term) = \langle\!\langle -k^{-1}, -\{x\}_k, +x \rangle\!\rangle$
Message: $M(x : Text \cup Nonce) = \langle\!\langle +x \rangle\!\rangle$
Key: $K(k : Key_P) = \langle\!\langle +k \rangle\!\rangle$
Hash: $H(x : Term) = \langle\!\langle -x, +h(x) \rangle\!\rangle$

Typically, $Key_P = \{key(x,y) \in Key \mid x = P \vee y = P\} \cup \{pvtkey(P)\} \cup \{pubkey(x) \mid x \in Name\}$.

2.2. History

A *history of protocol* Π is a tuple $h = \langle tr, \stackrel{msg}{\rightarrow}, role \rangle$, where tr is a trace mapping, $\stackrel{msg}{\rightarrow}$ is a binary relation on $\mathcal{N}_{tr}$, and $role \in \mathrm{dom}(tr) \rightarrow (\Pi \cup \Pi_P(Key_P))$ such that

1. For all $n_1, n_2 \in \mathcal{N}_{tr}$, if $n_1 \stackrel{msg}{\rightarrow} n_2$, then there exists $t \in Term$ such that $\mathrm{term}_{tr}(n_1) = +t$ and $\mathrm{term}_{tr}(n_2) = -t$. This represents that n_1 sends t, and n_2 receives t.
2. For all $n_1 \in \mathcal{N}_{tr}$, if $\mathrm{term}_{tr}(n_1)$ is negative, then there exists exactly one $n_2 \in \mathcal{N}_{tr}$ such that $n_2 \stackrel{msg}{\rightarrow} n_1$.
3. $\preceq_h$ is acyclic and well-founded (*i.e.*, does not have infinite descending chains), where $\preceq_h$ is the reflexive and transitive closure of $(\stackrel{msg}{\rightarrow} \cup \stackrel{tcl}{\rightarrow})$. Note that $\preceq_h$ is a partial order, first defined by Lamport [15].
4. For all $s \in \mathrm{dom}(tr)$, $tr(s)$ is a trace for $role(s)$. A *regular strand* is a strand s with $role(s) \in \Pi$. A *penetrator strand* is a strand s with $role(s) \in \Pi_P(Key_P)$. Nodes on regular and penetrator strands are called *regular nodes* and *penetrator nodes*, respectively. (For convenience, we assume $\Pi \cap \Pi_P(Key_P) = \emptyset$.)
5. For all $s \in \mathrm{dom}(tr)$, for all $x \in \mathrm{dom}(args(role(s), tr(s)))$, if parameter x is uniquely-originated, then $args(role(s), tr(s))(x)$ uniquely originates from $\langle s, i \rangle$, where i is the index of the first term in r that contains x.
6. For all $t \in \Pi.Secret$, t originates only from regular nodes.

Note that a history may contain multiple traces for the same role with identical bindings for parameters that are not uniquely originated.

To reduce clutter, we sometimes use a history instead of a trace mapping as a subscript; *e.g.*, for a history $h = \langle tr, \stackrel{msg}{\rightarrow}, role \rangle$, we define $\mathcal{N}_h = \mathcal{N}_{tr}$.

The set of *predecessors* of a node n in a history h is $\mathrm{preds}_h(n) = \{n' \in \mathcal{N}_h \mid n' \preceq_h n \wedge n' \neq n\}$.

Let $\mathrm{Hist}(\Pi)$ denote the set of histories of a protocol Π.

A set S of nodes is *backwards-closed* with respect to a binary relation R iff, for all nodes n_1 and n_2, if $n_2 \in S$ and $n_1 \; R \; n_2$, then $n_1 \in S$.

Given a history h of a protocol Π, a set S of nodes of h that is backward-closed with respect to $\preceq_h$ can be regarded as a history, denoted $\mathrm{nodesToHist}_h^{\Pi}(S)$, in a natural way.

2.3. Examples

Consider a payment protocol Π_{SET} based closely on [5] and reminiscent of SET [21], including the use of a dual-signature technique, so that the customer produces only one digital signature. Let $Order \subset Text$ and $PayDesc \subset Text$ denote sets of order and payment descriptions, respectively. Let $Price \subset Text$ and $Result \subset Text$ denote sets of prices and results (*e.g.*, "approved"), respectively. Let $Name_c$, $Name_m$, and $Name_g$ be disjoint subsets of $Name$ not containing P. For a set S of terms, let $Hash(S) = \{h(t) \mid t \in S\}$. The roles of protocol Π_{SET} appear in Figure 1, and $\Pi_{\mathrm{SET}}.Secret = \emptyset$, for reasons given below. We use **let** expressions to avoid repetition of large subterms. We allow $\mathrm{Cust}.m = P$ and $\mathrm{Gate}.m = P$ to model malicious merchants; similarly for malicious clients and gateways. There is no reason to allow the "me" variable of each role (namely, $\mathrm{Cust}.c$, $\mathrm{Mrch}.m$, and $\mathrm{Gate}.g$) to equal P, because P's actions are modeled by penetrator strands.

Use of $Hash(PayDesc)$ instead of the set of all hashes as the type for $\mathrm{Mrch}.hpd$ requires some justification, because a merchant cannot determine whether the hash received in *hpd* is the hash of a payment description or, say, a ciphertext. Attacks involving terms that are not of the expected type are called *type flaw attacks*. Use of the types $Hash(PayDesc)$ and $Hash(Order)$ can be justified by results like those in [12], which show that type flaw attacks can be prevented by using type tags in the protocol implementation. Extending their results to accommodate hashing and to accommodate the slightly larger class of agreement properties introduced below is fairly straightforward.

As another example, consider a version of the 1KP protocol [4] based closely on [8]. Following [8], we assume the customer account number (CAN) is secret and hence (for brevity) omit the PIN. We also omit the date field, since it does not affect the secrecy or agreement properties of Π_{1KP} given below, assuming nonces are uniquely-

$\text{Cust}(c : Name_c,\ m : Name_m \cup \{P\},\ g : Name_g \cup \{P\},\ \underline{nc} : Nonce,\ nm : Nonce,$
$\quad price : Price,\ od : Order,\ pd : PayDesc,\ result : Result) =$
let $trans = c{\cdot}m{\cdot}g{\cdot}nc{\cdot}nm{\cdot}price{\cdot}h(od){\cdot}h(pd)$ **in**
$\langle\!\langle +c{\cdot}m,$ (* 1. to merchant *)
$\ -nm,$ (* 2. from merchant *)
$\ +[trans]_{pvt(c)}{\cdot}\{od\}_{pub(m)}{\cdot}\{pd\}_{pub(g)},$ (* 3. to merchant *)
$\ -[result{\cdot}h(trans)]_{pvt(g)}\rangle\!\rangle$ (* 4. from gateway *)

$\text{Mrch}(c : Name_c \cup \{P\},\ m : Name_m,\ g : Name_g \cup \{P\},\ nc : Nonce,\ \underline{nm} : Nonce,$
$\quad price : Price,\ od : Order,\ hpd : Hash(PayDesc),\ epd : Term,\ result : Result) =$
let $trans = c{\cdot}m{\cdot}g{\cdot}nc{\cdot}nm{\cdot}price{\cdot}h(od){\cdot}hpd$ **in**
$\langle\!\langle -c{\cdot}m,$ (* 1. from customer *)
$\ +nm,$ (* 2. to customer *)
$\ -[trans]_{pvt(c)}{\cdot}\{od\}_{pub(m)}{\cdot}epd,$ (* 3. from customer *)
$\ +[trans]_{pvt(c)}{\cdot}[trans]_{pvt(m)}{\cdot}epd,$ (* 4. to gateway *)
$\ -[result{\cdot}h(trans)]_{pvt(g)}\rangle\!\rangle$ (* 5. from gateway *)

$\text{Gate}(c : Name_c \cup \{P\},\ m : Name_m \cup \{P\},\ g : Name_g,\ nc : Nonce,\ nm : Nonce,$
$\quad price : Price,\ hod : Hash(Order),\ pd : PayDesc,\ result : Result) =$
let $trans = c{\cdot}m{\cdot}g{\cdot}nc{\cdot}nm{\cdot}price{\cdot}hod{\cdot}h(pd)$ **in**
$\langle\!\langle -[trans]_{pvt(c)}{\cdot}[trans]_{pvt(m)}{\cdot}\{pd\}_{pub(g)}$ (* 1. from merchant *)
$\ +[result{\cdot}h(trans)]_{pvt(g)}\rangle\!\rangle$ (* 2. to merchant *)

Figure 1. Roles for Π_{SET}. Comments indicate step number and intended source or destination of message.

originated. Let $AcctNum \subset Text$ be a set of account numbers. To model dishonest customers (*i.e.*, customers that collude with the penetrator), we partition $AcctNum$ into two sets, $AcctNum_0$ and $AcctNum_1$, which contain account numbers of honest and dishonest customers, respectively. Let $Order$, $Result$, $Name_m$, and $Name_g$ be as above. We assume these subsets of $Text$ are disjoint. 1KP is designed for settings where the gateway has a private key with a well-known public key, but the customer and merchant do not. Consequently, 1KP provides few guarantees if the gateway is dishonest, so we do not include P in the types of Cust.g and Mrch.g. The roles of protocol $\Pi_{1\text{KP}}$ appear in Figure 2, and $\Pi_{1\text{KP}}.Secret = AcctNum_0$.

2.4. Derivability

Informally, a term t is derivable (by the pentrator) from a set S of nodes if the penetrator can compute t from $\text{term}(S)$ and Key_P. A formal definition follows.

For a nonce g that uniquely originates in a history h, let $\text{origin}_h(g)$ denote the node from which g originates in h.

For a set S of nodes in a history $h = \langle tr, \stackrel{msg}{\rightarrow}, role\rangle$ of a protocol Π, let $\text{uniqOrigReqrd}_h^{\Pi}(S)$ denote the set of nonces g such that there exists $\langle s, i\rangle \in S$ and $x \in \text{dom}(args(role(s), tr(s)))$ such that parameter x is uniquely originated and $args(role(s), tr(s))(x) = g$ and $\text{origin}_h(g) = \langle s, i\rangle$.

For a directed term t, the absolute value of t, denoted $\text{abs}(t)$, is t without its sign. For $T \subseteq Term$, $\text{abs}(T) = \{\text{abs}(t) \mid t \in T\}$, and the role Src_T is defined by $\text{Src}_T(x : T) = \langle\!\langle +x\rangle\!\rangle$.

A term t is *derivable* (by the penetrator) from a set S of nodes of a history h of a protocol Π, denoted $S \vdash_h^{\Pi} t$, if there exists a history $h' = \langle tr', \stackrel{msg'}{\rightarrow}, role'\rangle$ of the protocol $\{\text{Src}_{\text{abs}(\text{term}_h(S))}\}$ such that: (1) arguments of strands for Message in h' are not in $\text{uniqOrigReqrd}_h^{\Pi}(S)$; and (2) there exists a node $n \in \mathcal{N}_{tr'}$ with $\text{term}_{tr'}(n) = +t$. This relation is equivalent to the derivability relation in [7] and can be computed using the approach in [7].

2.5. Correctness Requirements

We consider the following kinds of correctness requirements. For a correctness requirement ϕ, we say that a protocol Π satisfies ϕ iff every history of Π satisfies ϕ.

Long-Term Secrecy. A history h of a protocol Π satisfies long-term secrecy iff, for every $t \in \Pi.Secret \cup (Key \setminus Key_P), \mathcal{N}_h \not\vdash_h^{\Pi} t$.

Cust($od : Order$, $price : Price$, $\underline{salt_c} : Nonce$, $\underline{R_c} : Nonce$, $CAN : AcctNum_0$,
$ID_m : Name_m \cup \{P\}$, $TID_m : Nonce$, $nonce_m : Nonce$, $g : Name_g$, $YesNo : Result$) =
let $cid = h(R_c \cdot CAN)$
and $common = price \cdot ID_m \cdot TID_m \cdot nonce_m \cdot cid \cdot h(od \cdot salt_c)$
and $clear = ID_m \cdot TID_m \cdot nonce_m \cdot h(common)$
and $slip = price \cdot h(common) \cdot CAN \cdot R_C$ **in**

$\langle\!\langle +salt_c \cdot cid,$	(* 1. to merchant *)
$-clear$	(* 2. from merchant *)
$+\{slip\}_{pub(g)},$	(* 3. to merchant *)
$-YesNo \cdot [h(YesNo \cdot h(common))]_{pvt(g)} \rangle\!\rangle$	(* 4. from merchant *)

Mrch($od : Order$, $price : Price$, $salt_c : Nonce$, $cid : Hash(Nonce \times AcctNum)$, $ID_m : Name_m$,
$TID_m : Nonce$, $\underline{nonce_m} : Nonce$, $g : Name_g$, $YesNo : Result$, $eslip : Term$) =
let $common = price \cdot ID_m \cdot TID_m \cdot nonce_m \cdot cid \cdot h(od \cdot salt_c)$
and $clear = ID_m \cdot TID_m \cdot nonce_m \cdot h(common)$ **in**

$\langle\!\langle -salt_c \cdot cid,$	(* 1. from customer *)
$+clear,$	(* 2. to customer *)
$-eslip,$	(* 3. from customer *)
$+clear \cdot h(od \cdot salt_c) \cdot eslip,$	(* 4. to gateway *)
$-YesNo \cdot [h(YesNo \cdot h(common))]_{pvt(g)},$	(* 5. from gateway *)
$+YesNo \cdot [h(YesNo \cdot h(common))]_{pvt(g)} \rangle\!\rangle$	(* 6. to customer *)

Gate($price : Price$, $R_c : Nonce$, $CAN : AcctNum$, $ID_m : Name_m \cup \{P\}$,
$TID_m : Nonce$, $nonce_m : Nonce$, $g : Name_g$, $hodsalt : Hash(Order \times Nonce)$, $YesNo : Result$) =
let $cid = h(R_c \cdot CAN)$
and $common = price \cdot ID_m \cdot TID_m \cdot nonce_m \cdot cid \cdot hodsalt$
and $clear = ID_m \cdot TID_m \cdot nonce_m \cdot h(common)$
and $slip = price \cdot h(common) \cdot CAN \cdot R_C$ **in**

$\langle\!\langle -clear \cdot hodsalt \cdot \{slip\}_{pub(g)},$	(* 1. from merchant *)
$+YesNo \cdot [h(YesNo \cdot h(common))]_{pvt(g)} \rangle\!\rangle$	(* 2. to merchant *)

Figure 2. Roles for Π_{1KP}.

Nonce Secrecy. Informally, nonce secrecy says: the values of specified nonce parameters are not revealed to the penetrator. A nonce secrecy requirement has the form "$r.x$ is secret unless $r.y \in S$", where $r \in \Pi$, x and y are parameters of r, and $S \subseteq Text$ (typically, $S \subseteq Name$). A history $h = \langle tr, \overset{msg}{\rightarrow}, role \rangle$ of a protocol Π satisfies that requirement iff, for every strand $s \in \mathrm{dom}(tr)$, if $role(s) = r$ and $y \in \mathrm{dom}(args(role(s), tr(s)))$ and $args(role(s), tr(s))(y) \notin S$, then $\mathcal{N}_{tr} \not\vdash_h^{\Pi} args(role(s), tr(s))(x)$.

Agreement. Informally, agreement says: if some strand executed a certain role to a certain point with certain arguments, then some strand must have executed a corresponding role to a corresponding point with corresponding arguments. An agreement requirement has the form "$\langle r_2, len_2 \rangle$ satisfying $x_2 \notin S_2$ is preceded by $\langle r_1, len_1 \rangle$ satisfying $t_1 = t_2$", where x_2 is a parameter of r_2, S_2 is a subset of $Text$, and t_1 and t_2 are terms containing parameters of r_1 and r_2, respectively, as free variables. A history $h = \langle tr, \overset{msg}{\rightarrow}, role \rangle$ of a protocol Π satisfies that agreement requirement iff, if h contains a strand s_2 such that $role(s_2) = r_2$, $\mathrm{len}(tr(s_2)) \geq len_2$, and $args(r_2, tr(s_2))(x_2) \notin S_2$, then tr contains a strand s_1 for role r_1 such that $\mathrm{len}(tr(s_1)) \geq len_1$ and t_1 instantiated with the arguments of s_1 equals t_2 instantiated with the arguments of s_2.

One of Bolignano's requirements for Π_{SET} is that the gateway has proof of transaction authorization by the merchant [5, p. 12]. This can be expressed as an agreement requirement: $\langle \mathrm{Gate}, 1 \rangle$ satisfying $\mathrm{Gate}.m \notin \{P\}$ is preceded by $\langle \mathrm{Mrch}, 4 \rangle$ satisfying

let $trans_m = \mathrm{Mrch}.c \cdot \mathrm{Mrch}.m \cdot \mathrm{Mrch}.nc \cdot \mathrm{Mrch}.nm$
$\cdot \mathrm{Mrch}.price \cdot h(\mathrm{Mrch}.od) \cdot \mathrm{Mrch}.hpd$
and $trans_g = \mathrm{Gate}.c \cdot \mathrm{Gate}.m \cdot \mathrm{Gate}.nc \cdot \mathrm{Gate}.nm$
$\cdot \mathrm{Gate}.price \cdot \mathrm{Gate}.hod \cdot h(\mathrm{Gate}.pd)$ **in**
$trans_m = trans_g \wedge \mathrm{Mrch}.g = \mathrm{Gate}.g$

This requirement applies even if Gate.$c = P$, *i.e.*, even if the customer is dishonest.[2] SET is designed to provide secrecy for order and payment descriptions. Π_{SET} as defined above does not provide such secrecy, because, *e.g.*, a customer strand with Cust.$m = P$ can reveal an order description to the penetrator. This is why we take $\Pi_{\mathrm{SET}}.\mathit{Secret} = \emptyset$. To express secrecy of order descriptions from gateways, we use a variant Π^{o}_{SET} in which merchants are assumed to be honest; specifically, Π^{o}_{SET} differs from Π_{SET} as follows: the type for Cust.m is Name_m, and $\Pi^{o}_{\mathrm{SET}}.\mathit{Secret} = \mathit{Order}$. Dishonest gateways are modeled by penetrator strands (the types of Cust.g and Mrch.g contain P), so if order descriptions are not known to the penetrator, then they are not known to dishonest gateways, so they are not known to honest gateways. Secrecy of payment descriptions from merchants can be expressed similarly.

Requirements for 1KP can be expressed similarly; for details, see [23]. 1KP also has a nonce secrecy requirement: Cust.R_c is secret unless Cust.$g \in \{P\}$.

3. Support

Informally, a set S' of nodes of a history tr supports a set S of nodes of tr if $S' \supseteq S$ and S' contains all of the regular nodes on which regular nodes in S depend. A formal definition follows.

For $T \subseteq \mathit{Term}$, the set of nonces that occur in T is $\mathrm{nonces}(T) = \{g \in \mathit{Nonce} \mid \exists t \in T : g \text{ occurs in } t\}$.

Let $\mathcal{RN}^{\Pi}_h$ denote the set of regular nodes in history h of protocol Π.

A set S' of nodes is a *support* for a set S of nodes in a history h of a protocol Π if:

1. $\mathcal{N}_h \supseteq S' \supseteq S$.
2. S' is backwards-closed with respect to $\overset{tcl}{\rightarrow}$.
3. For all negative nodes n in S', $\mathrm{preds}_h(n) \cap S' \cap \mathcal{RN}^{\Pi}_h \vdash^{\Pi}_h \mathrm{term}_h(n)$.
4. For all $g \in \mathrm{nonces}(\mathrm{term}_h(S')) \cap D$, where

   $D = \mathrm{uniqOrigReqrd}^{\Pi}_h(\mathcal{N}_h) \setminus \mathrm{uniqOrigReqrd}^{\Pi}_h(S')$,

   g occurs in the clear in $\mathrm{term}_h(\mathrm{origin}_h(g))$. (This condition ensures the compositionality property expressed in Lemma 2.)

For a strand s, if S' supports $\mathrm{nodes}(s)$, we say that S' supports s.

[2] Bolignano's version of the protocol omits g from *trans* and consequently violates the conjunct Mrch.g = Gate.g (in his presentation, this conjunct corresponds to $st'.\mathit{mcht}.\mathit{gateway} = G$ in the second filter function on p. 12).

For example, consider the following history of a generic payment protocol. Suppose $s_{c,1}$, $s_{m,1}$, and $s_{g,1}$ are customer, merchant, and gateway strands, respectively, that interact without interference from the penetrator. Let g be a nonce that uniquely originates on $s_{m,1}$ and is revealed to the penetrator (*e.g.*, the value of Mrch.nm in Π_{SET}). The penetrator then behaves as a merchant, interacting with a customer strand $s_{c,2}$ and a gateway strand $s_{g,2}$, except that the penetrator uses g instead of a fresh nonce. A support for $s_{c,2}$ or $s_{g,2}$ need not contain nodes on $s_{m,1}$ or $s_{c,1}$. In that sense, $s_{c,2}$ and $s_{g,2}$ do not depend on $s_{m,1}$, even though the chain of messages that conveys g means that there is causal dependence between those nodes in the classical sense of Lamport [15]. Informally, that classical dependence can be ignored here because the penetrator could generate a nonce g' and replace g with g' in the terms of nodes on $s_{c,2}$ and $s_{g,2}$. The careful treatment of unique origination in the definition of derivability allows such inessential classical dependencies to be ignored. The following lemma says that a support can be transformed into a history by adding penetrator nodes, without adding or changing regular nodes.

For a set S of nodes, let $\mathrm{strand}(S) = \{\mathrm{strand}(n) \mid n \in S\}$. For a trace mapping tr, a strand $s \in \mathrm{dom}(tr)$, and a set S of nodes of tr that is backwards-closed with respect to $\overset{tcl}{\rightarrow}$, S contains nodes on a prefix of $tr(s)$; let $\mathrm{prefix}_{tr}(s, S)$ denote that prefix.

Lemma 1. Let Π be a protocol. If S' is a support for S in a history $h = \langle tr, \overset{msg}{\rightarrow}, \mathit{role}\rangle$ of Π, then there exists a history $h' = \langle tr', \overset{msg'}{\rightarrow}, \mathit{role}'\rangle$ of Π such that

$$\begin{aligned}(\forall s \in \mathrm{strand}(S') : s \in \mathrm{dom}(tr') \wedge tr'(s) = \mathrm{prefix}_{tr}(s, S') \\ \wedge\ \mathit{role}'(s) = \mathit{role}(s)) \\ \wedge\ (\forall s \in \mathrm{dom}(tr') \setminus \mathrm{strand}(S') : \mathit{role}'(s) \in \Pi_P(\mathit{Key}_P)) \\ \wedge\ (\forall n_1, n_2 \in S' : n_1 \overset{msg'}{\rightarrow} n_2 \Rightarrow n_1 \overset{msg}{\rightarrow} n_2)\end{aligned} \tag{1}$$

Proof: h' is constructed by combining nodes in S with histories that witness the derivability of terms (as required by item 3 in the definition of support). For details, see [23]. ∎

Lemma 2. If S'_0 and S'_1 support S_0 and S_1, respectively, in a history $h = \langle tr, \overset{msg}{\rightarrow}, \mathit{role}\rangle$ of a protocol Π, then $S'_0 \cup S'_1$ supports $S_0 \cup S_1$ in history h of Π.

Proof: The only complication is dealing with nonces in $\mathrm{uniqOrigReqrd}^{\Pi}_h(S'_0) \setminus \mathrm{uniqOrigReqrd}^{\Pi}_h(S'_1)$ or $\mathrm{uniqOrigReqrd}^{\Pi}_h(S'_1) \setminus \mathrm{uniqOrigReqrd}^{\Pi}_h(S'_0)$. The fourth condition in the definition of support ensures that such nonces are available to the penetrator even if they are uniquely-originated. For details, see [23]. ∎

3.1. Bounded Support Restriction

A *strand count* for a protocol Π is a function from the roles of Π to the natural numbers. A set S of nodes *has strand count* f iff, for each role r, S contains nodes from exactly $f(r)$ strands for r. If $\mathcal{N}_h$ has strand count f, then we say that history h has strand count f. Let $f_1(r) = 1$ for every role r. We define a partial ordering $\preceq_{SC}$ on strand counts for a protocol; $\preceq_{SC}$ is simply the pointwise extension of the standard ordering on natural numbers.

A history h satisfies the *bounded support restriction*, abbreviated BSR, iff for each regular strand s in h, there exists a support for s in h with strand count at most f_1. A protocol satisfies BSR iff all of its histories do.

Π_{SET} and Π_{1KP} satisfy BSR. We proved these results manually; the proofs are similar to the proof in [22] for Lowe's corrected version of the Needham-Schroeder public-key authentication protocol. Theorem 2 in Section 5 shows that in principle, these results can be obtained automatically by state-space exploration of histories with bounded strand counts; an algorithm like the one in [22] can be used to compute a (small) support for a given set of nodes. The current bounds probably need to be decreased somewhat before this is feasible, *e.g.*, by finding a tighter bound on the dependence width (see Section 4).

4. Dependence Width

Informally, the dependence width of a negative term $r(i)$ in a role r of a protocol Π, denoted $\mathrm{DW}(\langle r, i\rangle, \Pi)$, is the maximum number of "additional" positive regular nodes needed in any history h of Π to provide the penetrator with enough knowledge to produce the term received by any node $\langle s, i\rangle$ of h such that $role(s) = r$. "Additional" here means "beyond those needed for the penetrator to produce negative terms that occur earlier in the same strand". The dependence width of a protocol Π, denoted $\mathrm{DW}(\Pi)$, is the maximum over all negative terms $r(i)$ in roles r in Π of $\mathrm{DW}(\langle r, i\rangle, \Pi)$. The concept of dependence width is used in the proof of Theorem 2 in Section 5 to bound the number of strands involved in a violation of BSR.

Let n be a negative node of a history h of a protocol Π, and let t be a subterm of $\mathrm{term}_h(n)$. A *revealing set* for t at n in h is a set S of positive regular nodes of tr such that $S \subseteq \mathrm{preds}_h(n)$ and $S \vdash_h^{\Pi} t$.

For a set S of numbers, let $\min(S)$ and $\max(S)$ denote the minimum and maximum element of S, respectively. We define $\min(\emptyset) = 0$ and $\max(\emptyset) = 0$.

The *revealing set min-size* of t at $\langle s, i\rangle$ in h is

$$\begin{aligned}&\mathrm{rvlSetMinSz}(t, \langle s, i\rangle, h) = \\ &\quad \min(\{\mathrm{size}(R \setminus \mathrm{nodes}_h(s)) \mid \\ &\qquad R \text{ is a revealing set for } t \text{ at } \langle s, i\rangle \text{ in } h\})\end{aligned} \tag{2}$$

Nodes in R that are on the same strand as n are not counted in the revealing set min-size (and hence not in the dependence width), because in the proof of Theorem 2—specifically, in equation (5)—those nodes appear in $\mathrm{support}_{h_0}^{\Pi}(s_0)$ and hence are excluded from the index set of the rightmost union, and the dependence width is designed to bound the size of that index set.

Note that, if there are no revealing sets for t at n in h (*i.e.*, t is not known to the penetrator at that point), then $\mathrm{rvlSetMinSz}(t, n, h) = 0$.

Let r be a role in a protocol Π, and let i be the index of a negative term in r. The *dependence width* of $\langle r, i\rangle$ in Π is

$$\begin{aligned}&\mathrm{DW}(\langle r, i\rangle, \Pi) = \\ &\max(\{\mathrm{rvlSetMinSz}(\mathrm{term}_{tr}(\langle s, i\rangle), \langle s, i\rangle, \langle tr, \stackrel{msg}{\rightarrow}, role\rangle) \mid \\ &\qquad \langle tr, \stackrel{msg}{\rightarrow}, role\rangle \in \mathrm{Hist}(\Pi) \wedge \langle s, i\rangle \in \mathcal{N}_{tr} \\ &\qquad \wedge\, role(s) = r\})\end{aligned} \tag{3}$$

The *dependence width* of a protocol Π is

$$\begin{aligned}&\mathrm{DW}(\Pi) = \max(\{\mathrm{DW}(\langle r, i\rangle, \Pi) \mid \\ &\qquad r \in \Pi \wedge r(i) \text{ is a negative term}\})\end{aligned} \tag{4}$$

The proof of Theorem 2, and therefore also the proof of Theorem 3, rely on an upper bound on the dependence width of the protocol. If the protocol might send terms of the forms $\{g\}_{k_1}$, $\{k_1\}_{k_2}$, $\{k_2\}_{k_3}$, ..., $\{k_{i-1}\}_{k_i}$, k_i, then $i + 1$ terms are needed to reveal g to the penetrator. Our long-term secrecy requirement prohibits such behavior. *Secrecy-limited dependence width*, abbreviated SL dependence width and denoted $\mathrm{DW}_{\mathrm{SL}}$, is defined in the same way as dependence width, except that the maximum over histories is restricted to histories satisfying long-term secrecy.

Let Π be a protocol, and let t be a term, possibly containing parameters. $\mathrm{nSecret}_0(t, \Pi)$ is a bound on the number of subterms of t that are not known to the penetrator, ignoring keys and values of parameters; formally, $\mathrm{nSecret}_0(t, \Pi) = N_c + N_h + N_{prim}$, where N_c is the number of subterms of t whose outermost operator is *encr*, ignoring those whose second argument is always in Key_P (based on parameter types), N_h is the number of subterms of t with outermost operator h, and N_{prim} is the number of elements of $Nonce \cup \Pi.Secret$ that occur in t. In computing N_c and N_h, identical subterms are counted only once. For a parameter $r.x$ of a role r of Π, $\mathrm{nSecret}(r.x, \Pi) = \max(\{\mathrm{nSecret}_0(t, \Pi) \mid t \text{ is in the type of } r.x\})$. Let $\mathrm{nSecret}(\langle r, i\rangle, \Pi) = \mathrm{nSecret}_0(r(i), \Pi) + \sum_{x \in \mathrm{params}(r(i))} \mathrm{nSecret}(r.x, \Pi)$, where $\mathrm{params}(t)$ is the set of parameters that occur in t.

Theorem 1. Let $r(i)$ be a negative term in a role r of a protocol Π. $\mathrm{DW}_{\mathrm{SL}}(\langle r, i\rangle, \Pi) \leq \mathrm{nSecret}(\langle r, i\rangle, \Pi)$.

Proof: Consider a strand s for r in a history h for Π. We consider each subterm t_1 of $\mathrm{term}_h(\langle s, i\rangle)$

and show that each hash, ciphertext, and element of $\text{uniqOrigReqrd}_h^{\Pi}(\mathcal{N}_h) \cup \Pi.Secret$ that occurs in $\text{term}_h(\langle s,i\rangle)$ contributes at most 1 to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$. The number of such subterms is bounded by $\text{nSecret}(\langle r,i\rangle,\Pi)$. Other subterms contribute nothing. The definition of dependence width implies that terms not derivable by the penetrator contribute nothing to the dependence width (because such terms have no revealing sets), so in computing the bound, we conservatively assume all subterms are derivable by the penetrator. Consider cases based on the type of t_1.

case 1: $t_1 \in Key$. Long-term secrecy implies that no keys are revealed, so keys contribute nothing to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$.

case 2: $t_1 \in \text{uniqOrigReqrd}_h^{\Pi}(\mathcal{N}_h) \cup \Pi.Secret$. The definition of history implies that t_1 originates from a regular node in h and (according to the conservative assumption discussed above) is derivable by the penetrator (using strands for Separation and Decryption), so there is a positive regular node n such that t_1 occurs in $\text{term}_h(n)$ either in the clear or encrypted only with keys known to the penetrator. Long-term secrecy implies that those keys (if any) are in Key_P. Thus, t_1 is derivable from $\{n\}$, so t_1 contributes at most 1 to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$.

case 3: $t_1 \in Text \setminus (\text{uniqOrigReqrd}_h^{\Pi}(\mathcal{N}_h) \cup \Pi.Secret)$. t_1 is directly available to the penetrator through the Message role, so t_1 contributes nothing to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$.

case 4: t_1 is a pair. Revealing a pair is equivalent to revealing its two components, so proper subterms of t_1 contribute to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$, but t_1 itself does not.

case 5: t_1 is a ciphertext or hash, and t_1 originates from a penetrator node in $\text{preds}_h(\langle s,i\rangle)$. The penetrator performs the encryption or hashing to construct its copy of t_1, so proper subterms of t_1 contribute to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$, but t_1 itself does not.

case 6: t_1 is a ciphertext or hash, and t_1 does not originate from a penetrator node in $\text{preds}_h(\langle s,i\rangle)$. Then t_1 originates from a regular node, and the argument is the same as in case 2. Note that it is not necessary for proper subterms of t_1 to contribute to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$. Our bound on $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$ might be loose because it does not attempt to exploit this observation; exploiting it is left for future work.

Now we justify ignoring, in the definition of N_c in nSecret_0, occurrences of *encr* whose second argument is always in Key_P. Let $\{t'\}_k$ be such a ciphertext.

case 1: $\emptyset \vdash_h^{\Pi} t'$; in other words, t' contains no secrets. Then $\emptyset \vdash_h^{\Pi} \{t'\}_k$, so $\{t'\}_k$ contributes nothing to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$.

case 2: $\emptyset \not\vdash_h^{\Pi} t'$; in other words, t' contains one or more secrets. Thus, subterms of t' contribute at least 1 to our bound on $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$.

> **case** 2.1: $\text{preds}_h(\langle s,i\rangle) \vdash_h^{\Pi} t'$. The penetrator can perform the encryption to construct its copy of $\{t'\}_k$, so proper subterms of $\{t'\}_k$ contribute to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$, but $\{t'\}_k$ itself does not, so ignoring $\{t'\}_k$ in N_c is safe.
>
> **case** 2.2: $\text{preds}_h(\langle s,i\rangle) \not\vdash_h^{\Pi} t'$. The ciphertext $\{t'\}_k$ must originate from a regular node and be revealed to the penetrator. The ciphertext actually contributes 1 to $\text{DW}_{\text{SL}}(\langle r,i\rangle,\Pi)$ (*cf.* case 6 above), and its subterms actually contribute nothing. Our bound counts 0 from the ciphertext but counts at least 1 from subterms of t'. Thus, although the bookkeeping might seem skewed, the sum of the contributions is sufficient. ∎

We simplify Π_{SET} and Π_{1KP} as follows. Parameters *epd* and *eslip* are used to forward messages in a trivial way (specifically, all occurrences of these parameters are unencrypted), and TID_m is redundant because it always appears together with $nonce_m$. Thus, eliminating these parameters has no impact on correctness. Let Π'_{SET} and Π'_{1KP} refer to versions of the protocols in which these parameters have been eliminated. Theorem 1 implies $\text{DW}_{\text{SL}}(\Pi'_{\text{SET}}) \leq 6$ and $\text{DW}_{\text{SL}}(\Pi'_{\text{1KP}}) \leq 7$. In both protocols, the first term of Gate has the largest dependence width.

The bound on DW_{SL} provided by Theorem 1 can sometimes be decreased by replacing a negative term of the form $-t_1 \cdot t_2$ in a role with the sequence of terms $-t_1, -t_2$. For example, let Π''_{SET} denote the protocol obtained from Π'_{SET} by splitting the first term of Gate into a sequence of three terms. Theorem 1 implies $\text{DW}_{\text{SL}}(\Pi''_{\text{SET}}) \leq 5$. This transformation preserves all correctness requirements, provided the lengths in agreement requirements are adjusted appropriately.

5. Reduction for BSR and Long-Term Secrecy

The following lemma says, roughly, that constructing a history h' from a support S' of a set S of nodes of a history h does not create new supports for S.

Lemma 3. Suppose S_0 supports S in a history h of a protocol Π. Let h' be a history of Π whose existence is implied by Lemma 1 applied to S_0. Suppose S_1 supports S in history h' of Π. Then $S_1 \cap \mathcal{RN}_h^{\Pi}$ supports S in history h of Π.

Proof: The proof is similar to that of Lemma 3 in [22]. ∎

For a protocol Π, define a strand count $\beta(\Pi)$ by $\beta(\Pi)(r) = \text{DW}_{\text{SL}}(\Pi) + 1$.

Theorem 2. A protocol Π satisfies BSR and long-term secrecy iff all histories of Π with strand count $\beta(\Pi)$ do.

Proof: The forward direction ($\Rightarrow$) of the "iff" is easy. For the reverse direction ($\Leftarrow$), we prove the contrapositive, *i.e.*, we suppose there exists a history h of Π that violates BSR or long-term secrecy, and we construct a history of Π with strand count at most $\beta(\Pi)$ that violates the same property.

BSR and long-term secrecy are safety properties satisfied by histories with zero nodes, and $\preceq_h$ is well-founded, so there exists a $\preceq_h$-minimal node n_0 such that

1. $\text{nodesToHist}_h^{\Pi}(\text{preds}_h(n_0))$ satisfies BSR and long-term secrecy.
2. $\text{nodesToHist}_h^{\Pi}(\text{preds}_h(n_0)) \cup \{n_0\}$ violates BSR or long-term secrecy.

Let $h_0 = \text{nodesToHist}_h^{\Pi}(\text{preds}_h(n_0))$. Let $s_0 = \text{strand}(n_0)$ and $i_0 = \text{index}(n_0)$. Note that in h_0, s_0 does not include n_0. For a strand s in a history h' that satisfies BSR, let $\text{support}_{h'}(s)$ denote a support for s in h' with strand count at most f_1. The definitions of BSR and long-term secrecy imply n_0 is a regular node. Consider cases based on the sign of n_0.

case: n_0 is a negative node. n_0 cannot cause a violation of secrecy, so it causes a violation of BSR. Suppose $i_0 > 0$. n_0 directly depends on $\langle s_0, i_0 - 1\rangle$ and on a revealing set R for $\text{term}(n_0)$ at n_0 in h; more precisely, for all S', if S' supports $\{\langle s_0, i_0 - 1\rangle\} \cup R$ in h, then $S' \cup \{n_0\}$ supports $\{n_0\}$ in h. h_0 satisfies long-term secrecy, so Theorem 1 implies $\text{size}(R \setminus \text{nodes}_{h_0}(s_0)) \leq \text{DW}_{\text{SL}}(\Pi)$. Let

$$S_1 = \{n_0\} \cup \text{support}_{h_0}(s_0) \cup \bigcup_{n \in R \setminus \text{nodes}_{tr_0}(s_0)} \text{support}_{h_0}(\text{strand}(n)). \quad (5)$$

h_0 satisfies BSR, so each of the supports in (5) has strand count at most f_1, so S_1 has strand count at most $\beta(\Pi)$ (note that n_0 is on s_0, so $\{n_0\} \cup \text{support}_{h_0}(s_0)$ contributes at most f_1 to the strand count of S_1).

Lemma 2 implies that $S_1 \setminus \{n_0\}$ supports $\{\langle s_0, i_0 - 1\rangle\} \cup R$ in h; thus, S_1 supports $\{n_0\}$ in h. Lemma 1 implies that S_1 can be transformed into a history h_1 of Π by adding penetrator nodes. Adding penetrator nodes does not affect the strand count, so h_1 has strand count at most $\beta(\Pi)$. We show by contradiction that n_0 also causes a violation of BSR in h_1. Suppose n_0 does not cause such a violation. Then there exists a support S' for $\{n_0\}$ in h_1 with strand count at most f_1. Lemma 3 implies that $S' \cap \mathcal{RN}_{h_1}^{\Pi}$ is a support for $\{n_0\}$ in h with strand count at most f_1, a contradiction.

Suppose $i_0 = 0$. The proof is similar to the case $i_0 > 0$, except n_0 does not depend on the non-existent node $\langle s_0, i_0 - 1\rangle$, so we omit $\text{support}_{h_0}(s_0)$ from the definition of S_1, and Lemma Lemma 2 implies that $S_1 \setminus \{n_0\}$ supports R in h.

case: n_0 is a positive node. n_0 cannot cause a violation of BSR, so it causes a violation of long-term secrecy. $\text{preds}_h(n_0)$ satisfies long-term secrecy, so there is some $t \in \Pi.\mathit{Secret} \cup (\mathit{Key} \setminus \mathit{Key}_P)$ such that t appears in $\text{term}_h(n_0)$ either in the clear or encrypted only with keys in Key_P. Suppose $i_0 > 0$. Let $S_0 = \text{support}_{h_0}(s_0)$ and $S_1 = \{n_0\} \cup S_0$. h_0 satisfies BSR, so S_0 and S_1 have strand count at most f_1 (note that n_0 is on s_0, and $s_0 \in \text{strand}(S_0)$, so n_0 does not increase the strand count of S_1). S_1 can be transformed into a history h_1 by adding penetrator nodes; this follows from Lemma 1 and the observation that n_0 is positive and is an immediate successor of the last node on s_0 in h_0. It is easy to show that adding penetrator nodes does not change the strand count or destroy the violation of long-term secrecy. Thus, h_1 is a history of Π with strand count at most $\beta(\Pi)$ that violates long-term secrecy. Suppose $i_0 = 0$. Then $\text{preds}_h(n_0) = \emptyset$, and the history containing only node n_0 has strand count at most f_1 and violates long-term secrecy. ∎

6. Reduction for Nonce Secrecy and Agreement

Define a strand count f_2 by: $f_2(r) = 2$ for every role r.

Theorem 3. Let ϕ be a nonce secrecy or agreement requirement. Suppose all histories of a protocol Π with strand count $\beta(\Pi)$ satisfy BSR and long-term secrecy. Π satisfies ϕ iff all histories of Π with strand count f_2 do.

Proof: The forward direction ($\Rightarrow$) of the "iff" is easy. For the reverse direction ($\Leftarrow$), we prove the contrapositive, *i.e.*, we suppose there exists a history $h = \langle tr, \overset{msg}{\rightarrow}, role\rangle$ of Π that violates ϕ, and we construct a history of Π with strand count at most f_2 that violates ϕ. Nonce secrecy and agreement requirements are safety properties satisfied by histories with zero nodes, and $\preceq_h$ is well-founded, so there exists a $\preceq_h$-minimal node n_0 such that

1. $\text{nodesToHist}_h^{\Pi}(\text{preds}_h(n_0))$ satisfies ϕ.
2. $\text{nodesToHist}_h^{\Pi}(\text{preds}_h(n_0)) \cup \{n_0\}$ violates ϕ.

Let $s_0 = \text{strand}(n_0)$.

By hypothesis, all histories of Π with strand count $\beta(\Pi)$ satisfy BSR and long-term secrecy, so Theorem 2 implies that Π satisfies BSR. For $s \in \text{dom}(h)$, let $\text{support}_h(s)$ denote a support for s with strand count at most f_1.

Suppose ϕ is a nonce secrecy requirement. ϕ has the form "$r.x$ is secret unless $r.y \in S$". n_0 is a positive regular node, and there is a regular strand s_g such that $\mathit{args}(\mathit{role}(s_g), \mathit{tr}(s_g))(y) \notin S$ and $\text{preds}_h(n_0) \not\vdash_h^{\Pi} g$ and $\text{preds}_h(n_0) \cup \{n_0\} \vdash_h^{\Pi} g$, where $g = \mathit{args}(\mathit{role}(s), \mathit{tr}(s))(x)$. By the same reasoning as in case

2 of the proof of Theorem 1, this implies that $\{n_0\} \vdash_h^{\Pi} g$. Let $S_1 = \text{support}_h(s_0) \cup \text{support}_h(s_g)$. Lemma 2 implies that S_1 is a support for $\text{nodes}_h(s_0) \cup \text{nodes}_h(s_g)$. Lemma 1 implies that S_1 can be transformed into a history h_1 by adding penetrator nodes. Note that S_1 and h_1 have strand count at most f_2. It is easy to see that n_0 causes a violation of nonce secrecy in h_1.

Suppose ϕ is an agreement requirement. ϕ has the form: "$\langle r_2, len_2\rangle$ satisfying $x_2 \notin S_2$ is preceded by $\langle r_1, len_1\rangle$ satisfying $t_1 = t_2$". n_0 causes a violation of ϕ, so s_0 is a strand for r_2 and $args(r_2, tr(s_2))(x_2) \notin S_2$ and $\text{index}(n_0) = len_2$. Lemma 1 implies that $\text{support}_h(s_0)$ can be transformed into a history h_0 of Π with strand count at most f_1. Note that $n_0 \in \mathcal{N}_{h_0}$. Removing nodes in $\mathcal{N}_h \setminus \mathcal{N}_{h_0}$ and adding penetrator nodes preserve the lack of a node $\langle s_1, len_1\rangle$ such that $role(s_1) = r_1$ and such that t_1 instantiated with the arguments of s_1 equals t_2 instantiated with the arguments of s_0. Thus, h_0 violates ϕ. ∎

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 143:1–70, 1999.

[2] P. A. Abdulla and B. Jonsson. Verifying networks of timed processes. In *Proc. 4th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 1998.

[3] K. Baukus, K. Stahl, S. Bensalem, and Y. Lakhnech. Abstracting ws1s systems to verify parameterized networks. In *Proc. 6th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 188–203. Springer-Verlag, 2000.

[4] M. Bellare, J. A. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, E. V. Herreweghen, and M. Waidner. Design, implementation and deployment of a secure account-based electronic payment system. *IEEE Journal on Selected Areas in Communications*, 18(4):611–627, 2000.

[5] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 1997.

[6] D. Bolignano. Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In A. J. Hu and M. Y. Vardi, editors, *Proc. Tenth Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[7] E. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, June 1998.

[8] E. Clarke, W. Marrero, and S. Jha. A machine checkable logic of knowledge for specifying security properties of electronic commerce protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, June 1998.

[9] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstractions and regular languages. In *Proc. Sixth Int'l. Conference on Concurrency Theory (CONCUR)*, 1995.

[10] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proc. 1999 Workshop on Formal Methods and Security Protocols*, July 1999. Available via http://cm.bell-labs.com/cm/cs/who/nch/fmsp99/.

[11] E. A. Emerson and K. S. Namjoshi. Automated verification of parameterized synchronous systems. In *Proc. 8th Int'l. Conference on Computer-Aided Verification (CAV)*, 1996.

[12] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proc. 13th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2000.

[13] N. Heintze, J. D. Tygar, J. Wing, and H.-C. Wong. Model checking electronic commerce protocols. In *Proc. USENIX 1996 Workshop on Electronic Commerce*, 1996.

[14] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite state systems. In *Proc. 6th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 220–234. Springer-Verlag, 2000.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[16] G. Lowe. Towards a completeness result for model checking of security protocols. *The Journal of Computer Security*, 7(2/3):89–146, 1999.

[17] D. Marchignoli and F. Martinelli. Automatic verification of cryptographic protocols through compositional analysis techniques. In *Proc. 5th Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 148–162. Springer-Verlag, 1999.

[18] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[19] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Seventh USENIX Security Symposium*, pages 201–216, 1998.

[20] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *The Journal of Computer Security*, 7(2/3), 1999.

[21] SET: Secure Electronic Transaction Specification, version 1.0, May 1997. Available from www.setco.org.

[22] S. D. Stoller. A bound on attacks on authentication protocols. Technical Report 526, Computer Science Dept., Indiana University, July 1999. Submitted for journal publication.

[23] S. D. Stoller. A bound on attacks on payment protocols. Technical Report 537, Computer Science Dept., Indiana University, February 2000. Revised April 2001. Available via http://www.cs.sunysb.edu/~stoller/ .

[24] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 18th IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1998.

A Dichotomy in the Complexity of Propositional Circumscription

Lefteris M. Kirousis*
Computer Engineering and Informatics
University of Patras
GR-265 04 Patras, Greece.
kirousis@ceid.upatras.gr

Phokion G. Kolaitis†
Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064, U.S.A.
kolaitis@cse.ucsc.edu

Abstract

The inference problem for propositional circumscription is known to be highly intractable and, in fact, harder than the inference problem for classical propositional logic. More precisely, in its full generality this problem is Π_2^P-complete, which means that it has the same inherent computational complexity as the satisfiability problem for quantified Boolean formulas with two alternations (universal-existential) of quantifiers. We use Schaefer's framework of generalized satisfiability problems to study the family of all restricted cases of the inference problem for propositional circumscription. Our main result yields a complete classification of the "truly hard" (Π_2^P-complete) and the "easier" cases of this problem (reducible to the inference problem for classical propositional logic). Specifically, we establish a dichotomy theorem which asserts that each such restricted case either is Π_2^P-complete or is in coNP. *Moreover, we provide efficiently checkable criteria that tell apart the "truly hard" cases from the "easier" ones.*

1 Introduction and Summary of Results

During the past three decades, researchers in artificial intelligence have investigated in depth various formalisms of nonmonotonic reasoning. Circumscription, introduced by McCarthy [McC80], is perhaps the most well-known and extensively studied such formalism. It enjoys high expressive power and thus is suitable for modeling a wide variety of problems requiring nonmonotonic reasoning. Moreover, propositional circumscription has been shown by Gelfond et al. [GPP89] to coincide with reasoning under the extended closed world assumption (ECWA), which is one of the main formalisms for reasoning with incomplete information.

*Research partially supported by the Research Committee of the University of Patras and by the Computer Technology Institute.

†Research partially supported by NSF Grants No. CCR-9610257 and No. CCR-9732041

A fundamental problem in every logical formalism is *inference*, i.e., the problem of deciding whether, given two formulas φ and ψ, the formula ψ can be inferred from φ in the context of the logical formalism at hand. Intuitively, φ represents a knowledge base, while ψ represents a statement that we are interested in deciding whether it can be inferred from the knowledge base. In the case of classical propositional logic, inference amounts to *tautological implication* $\varphi \models \psi$, i.e., to the problem of deciding whether ψ is satisfied by every truth assignment that satisfies φ. Consequently, inference in classical propositional logic is a coNP-complete problem and thus considered to be intractable. In the case of propositional circumscription, inference turns out to have even higher inherent computational complexity. Indeed, as shown by Eiter and Gottlob [EG93], the inference problem for propositional circumscription is Π_2^P-complete. Recall that the class Π_2^P constitutes the second level of the polynomial hierarchy PH and thus contains both NP and coNP as subclasses. Moreover, the prototypical Π_2^P-complete problem is Π_2^P-SAT, i.e., the satisfiability problem for quantified Boolean formulas of the form $\forall\bar{x}\exists\bar{y}\theta(\bar{x},\bar{y})$, where $\bar{x}$, $\bar{y}$ are tuples of propositional variables and $\theta(\bar{x},\bar{y})$ is a CNF-formula (see [Pap94]).

Classical propositional logic is concerned with all models of a given formula, i.e., with all truth assignments that satisfy the formula. In contrast, propositional circumscription is concerned with the *minimal* models of a given formula, i.e., with those satisfying truth assignments for which there is no smaller satisfying truth assignment with respect to the coordinate-wise partial order between truth assignments. Consequently, in its full generality, the inference problem for propositional circumscription can be stated as follows: given two CNF-formulas φ and ψ, is ψ true in every minimal model of φ? A moment's reflection reveals that this problem is polynomial-time equivalent to the special case in which ψ is simply a clause (i.e., a disjunction of literals), since ψ can be inferred from φ under propositional circumscription if and only if each clause of ψ can be so inferred. Moreover, Eiter and Gottlob [EG93] established that

0-7695-1281-X/01 $10.00 © 2001 IEEE

the inference problem for propositional circumscription remains Π_2^P-complete even when φ is a 3CNF-formula and the clause ψ consists of a single negated variable.

Are there restricted classes of propositional formulas on which the inference problem for propositional circumscription has complexity lower than Π_2^P-complete? To make this question precise, one can consider restrictions on both the formulas representing knowledge bases and the formulas representing statements to be inferred. Since clauses are the syntactically simplest propositional formulas, it is natural to consider restrictions on the formulas representing knowledge bases only. Thus, for every class $\mathcal{F}$ of propositional formulas, we let INF-CIRC($\mathcal{F}$) denote the following decision problem: given a formula $\varphi \in \mathcal{F}$ and a clause ψ, is ψ true on every minimal model of φ? The question then is to analyze the computational complexity of INF-CIRC($\mathcal{F}$) for different classes $\mathcal{F}$ of propositional formulas and identify classes $\mathcal{F}$ for which the complexity of INF-CIRC($\mathcal{F}$) is lower than Π_2^P-complete. Even before the Π_2^P-completeness of the full problem was established, this question was studied by Cadoli and Lenzerini [CL94], where INF-CIRC($\mathcal{F}$) was shown to be in P or to be coNP-complete for several different classes $\mathcal{F}$ of propositional formulas. Specifically, Cadoli and Lenzerini observed that if a class $\mathcal{F}$ of propositional formulas is such that testing satisfying truth assignments for minimality is in polynomial time, then INF-CIRC($\mathcal{F}$) is in coNP. Since minimality testing is in polynomial time for the classes of Horn formulas, dual Horn formulas and 2CNF-formulas, it follows that INF-CIRC($\mathcal{F}$) is in coNP, when $\mathcal{F}$ is one of these three classes. Moreover, if $\mathcal{F}$ is the class of all Horn formulas, then INF-CIRC($\mathcal{F}$) is solvable in polynomial time, since every satisfiable Horn formula has a minimum (unique minimal) model that can be computed in polynomial time. In [CL94], it was also proved that INF-CIRC($\mathcal{F}$) is actually coNP-complete, when $\mathcal{F}$ is the class of all dual Horn formulas or the class of all 2CNF-formulas.

The aforementioned results identify several interesting cases where the complexity of the inference problem in propositional circumscription is lower than Π_2^P-complete. Nonetheless, they do not provide a *complete* classification of the "truly hard" (Π_2^P-complete) and the "easier" cases of this problem. In particular, except for the class of all CNF-formulas and the class of all 3CNF-formulas, no other interesting classes $\mathcal{F}$ of propositional formulas for which INF-CIRC($\mathcal{F}$) is Π_2^P-complete were known prior to the work reported here. This should be contrasted with the state of affairs concerning the complexity of the inference problem for classical propositional logic, where a complete classification can be derived from the pioneering work by Schaefer [Sch78] on the complexity of GENERALIZED SATISFIABILITY problems. In order to describe Schaefer's work and relate it to the inference problem, we need to introduce some terminology and notation.

A *logical relation* (or *generalized connective*) R is a non-empty subset of $\{0,1\}^k$, for some $k \geq 1$. If $S = \{R_1, \ldots, R_m, \ldots\}$ is a set of logical relations, then an $\mathcal{F}(S)$-formula is a conjunction of expressions (called *generalized clauses* or, simply, *clauses*) of the form $\mathbf{R}_i(x_1, \ldots, x_k)$, where each $\mathbf{R}_i$ is a relation symbol representing the logical relation R_i in S and each x_j is a Boolean variable. Furthermore, an $\mathcal{F}_C(S)$-formula is a formula obtained from an $\mathcal{F}(S)$-formula by substituting some of the variables by the constant symbols $\mathbf{0}$ and $\mathbf{1}$. Each set S of logical relations gives rise to the following GENERALIZED SATISFIABILITY problem SAT$_C(S)$: given an $\mathcal{F}_C(S)$-formula φ, is φ satisfiable? In a similar manner, one obtains the family of SAT(S) problems by considering $\mathcal{F}(S)$-formulas, instead of $\mathcal{F}_C(S)$-formulas.

In [Sch78], four conditions were isolated and the following remarkable classification theorem for the family of all GENERALIZED SATISFIABILITY problems SAT$_C(S)$ was established: if the set S satisfies at least one of these four conditions, then SAT$_C(S)$ is solvable in polynomial time; otherwise, SAT$_C(S)$ is NP-complete. These four conditions are: (1) every relation in S is the set of models of a Horn formula; (2) every relation in S is the set of models of a dual Horn formula; (3) every relation in S is the set of models of a 2CNF formula; (4) every relation in S is the set of models of an affine formula, i.e., a conjunction of formulas built using the $\oplus$ (exclusive or) connective. It should be noted that each of these conditions turned out to be efficiently checkable. Schaefer also obtained a classification theorem for the family of SAT(S) problems, which involves two additional conditions that trivially give rise to polynomial-time solvable SAT(S) problems. Note that the NP-completeness of POSITIVE 1-IN-3-SAT, NOT-ALL-EQUAL 3-SAT and other well known variants of SAT is an immediate consequence of Schaefer's results. Morover, the above results constitute the first instance of a *dichotomy theorem* for a family of decision problems in NP, i.e., results that concern an infinite family $\mathcal{C}$ of decision problems and assert that certain problems in $\mathcal{C}$ are NP-complete, while on the contrary all other problems in $\mathcal{C}$ are solvable in polynomial time. It should be pointed out that the a priori existence of dichotomy theorems cannot be taken for granted, since Ladner's theorem in [Lad75] asserts that if P $\neq$ NP, then there are problems in NP that are neither NP-complete nor in P.

The inference problem in classical propositional logic is polynomial-time reducible to the satisfiability problem. Using this fact, it is easy to see that Schaefer's dichotomy theorem for satisfiability problems yields a dichotomy theorem for the inference problem in classical propositional logic. Specifically, if S is a set of logical relations that satisfy at least one of the four aforementioned conditions, then the inference problem in classical propositional logic

for $\mathcal{F}_{\mathrm{C}}(S)$-formulas is solvable in polynomial time; otherwise, it is coNP-complete. In addition, a similar dichotomy theorem can be derived for the inference problem in classical propositional logic for $\mathcal{F}(S)$-formulas.

In this paper, we use Schaefer's framework to investigate the computational complexity of the inference problem in propositional circumscription. Our main result asserts that, for every set S of logical relations, either INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is Π_2^{P}-complete or INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is in coNP. In other words, our main result tells that each restricted cases of the inference problem for propositional circumscription either is as hard as the general case or is reducible to the inference problem for classical propositional logic. Moreover, it provides efficiently checkable criteria that, given a finite set S of logical relations, distinguish the two possibilities for the complexity of INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$). This constitutes a dichotomy theorem for the inference problem in propositional circumscription, since results by Ladner [Lad75] imply that if $\Pi_2^{\mathrm{P}} \neq$ coNP, then there are decision problems in Π_2^{P} that are neither Π_2^{P}-complete nor in coNP. It should also be pointed out that the boundary in the dichotomy separating Π_2^{P}-completeness from membership in coNP turns out to be different from the boundary in the dichotomy theorem for the inference problem in classical propositional logic.

Our main result is established in two stages. In the first stage, we prove a dichotomy theorem for the family of INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) problems, where S is a set of 1-*valid* logical relations, i.e., each relation in S contains the all-ones tuple $(1, \ldots, 1)$. In the second stage, we use this restricted dichotomy theorem as a stepping stone to derive the dichotomy theorem for the full family of INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) problems, where S is an arbitrary set of logical relations. To this effect, we apply the restricted dichotomy theorem to the set S^* of all 1-valid logical relations obtained from relations in S by replacing some variables by 0. A two-stage approach was used for the first time in a recent paper [KK01], where a dichotomy theorem for minimal satisfiability problems was established. With some extra work, we can also obtain a dichotomy theorem for the family of all INF-CIRC($\mathcal{F}(S)$) problems, where S is a set of logical relations. Due to space limitations, this result will be presented in the full version of the present paper.

Since the publication of the original dichotomy theorem by Schaefer [Sch78], researchers have obtained several other dichotomy theorems for certain variants of satisfiability problems (see, for instance, [Cre95, KSW97, CH96, CH97, KS98, RV00, KK01]). The results reported here provide the first dichotomy between Π_2^{P}-completeness and membership in coNP. At the technical level, the proofs make extensive use of Schaefer's expressibility theorem [Sch78, Theore 3.0], as well as of a definability result by Creignou and Hébrard [CH97] and other special-purpose definability results established here.

Finally, we conjecture that a *trichotomy* theorem holds for the complexity of propositional circumscription. Specifically, we conjecture that, for every set S of logical relations, exactly one of the following three alternatives holds: (1) INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is Π_2^{P}-complete; (2) INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is coNP-complete; (3) INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is solvable in polynomial time. Note that if this conjecture is confirmed, it will yield the first trichotomy theorem for a family of natural decision problems in a complexity class beyond NP. In view of the dichotomy theorem established here, it remains to establish a dichotomy theorem for those INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) problems that are in coNP. Although the results in [CL94] yield parts of this conjectured dichotomy, much more remains to be done in order to complete the picture.

2 Preliminaries and Background

This section contains a minimum amount of the necessary background material on the complexity of GENERALIZED SATISFIABILITY problems from [Sch78].

Let $S = \{R_1, \ldots, R_m, \ldots\}$ be a set of logical relations of various arities. As stated in Section 1, an $\mathcal{F}(S)$-formula is a finite conjunction of clauses built using relations from S and propositional variables, while an $\mathcal{F}_{\mathrm{C}}(S)$-formula is a formula built using relations from S, propositional variables, and the constant symbols **0** or **1**. Recall also that SAT(S) is the following decision problem: given an $\mathcal{F}(S)$-formula φ, is it satisfiable? (i.e., is there a truth assignment to the variables of φ that makes every clause of φ true?) The decision problem SAT$_{\mathrm{C}}(S)$ is defined in a similar way.

Clearly, for each finite set S of logical relations, both SAT(S) and SAT$_{\mathrm{C}}(S)$ are problems in NP. Several well-known NP-complete problems can easily be cast as SAT(S) problems for particular sets S of logical relations. For example, 3-SAT coincides with the problem SAT(S), where $S = \{R_0, R_1, R_2, R_3\}$ and $R_0 = \{0,1\}^3 - \{(0,0,0)\}$ (expressing the clause $(x \vee y \vee z)$), $R_1 = \{0,1\}^3 - \{(1,0,0)\}$ (expressing the clause $(\neg x \vee y \vee z)$), $R_2 = \{0,1\}^3 - \{(1,1,0)\}$ (expressing the clause $(\neg x \vee \neg y \vee z)$), and $R_3 = \{0,1\}^3 - \{(1,1,1)\}$ (expressing the clause $(\neg x \vee \neg y \vee \neg z)$). Similarly, the NP-complete problem POSITIVE-1-IN-3-SAT ([GJ79, LO4, page 259]) is precisely the problem SAT(S), where S is the singleton consisting of the relation $R_{1/3} = \{(1,0,0), (0,1,0), (0,0,1)\}$.

Recall that a *Horn* formula is a conjunction of clauses each of which is a disjunction of literals such that at most one of them is a variable. Similarly, a *dual Horn* formula is a conjunction of clauses each of which is disjunction of literals such that at most one of them is a negated variable. As mentioned in Section 1, an *affine* formula is a conjunction of subformulas each of which is an *exclusive disjunction* $\oplus$ of

literals or a negation of an exclusive disjunction of literals.

Definition 2.1: Let R be a logical relation and S a finite set of logical relations.

R is 1-*valid* if it contains the tuple $(1, 1, \ldots, 1)$, whereas R is 0-*valid* if it contains the tuple $(0, 0, \ldots, 0)$. We say that S is 1-*valid* (0-*valid*) if every member of S is 1-valid (0-valid).

R is *2CNF* (*Horn*, *dual Horn*, or *affine*, respectively) if there is a propositional formula φ which is 2CNF (Horn, dual Horn, or affine, respectively) and such that R coincides with the set of truth assignments satisfying φ.

S is *Schaefer* if at least one of the following four conditions hold: every member of S is 2CNF; every member of S is Horn; every member of S is dual Horn; every member of S is affine. Otherwise, we say that S is *non-Schaefer*. ∎

There are efficient criteria to determine whether a logical relation is 2CNF, Horn, dual Horn, or affine. In fact, a set of such criteria was already provided by Schaefer [Sch78]; moreover, even simpler criteria for a relation to be Horn or dual Horn were given by Dechter and Pearl [DP92]. Each of these criteria involves a *closure property* of the logical relations at hand under a certain function. Specifically, a relation R is 2CNF if and only if for all $t_1, t_2, t_3 \in R$, we have that $(t_1 \vee t_2) \wedge (t_2 \vee t_3) \wedge (t_1 \vee t_3) \in R$, where the operators $\vee$ and $\wedge$ are applied coordinate-wise to bit tuples. R is Horn (respectively, dual Horn) if and only if for all $t_1, t_2 \in R$, we have that $t_1 \wedge t_2 \in R$ (respectively, $t_1 \vee t_2 \in R$). Finally, R is affine if and only if for all $t_1, t_2, t_3 \in R$, we have that $t_1 \oplus t_2 \oplus t_3 \in R$.

If S is a 0-valid or a 1-valid set of logical relations, then $\text{SAT}(S)$ is a trivial decision problem (the answer is always "yes"). If S is an affine set of logical relations, then $\text{SAT}(S)$ can be solved in polynomial time using Gaussian elimination. Moreover, there are well-known polynomial-time algorithms for the satisfiability problem for the class of all 2CNF formulas (2-SAT), the class of all Horn formulas, and the class of all dual Horn formulas. Schaefer's seminal discovery was that the above six cases are the *only* tractable cases of $\text{SAT}(S)$; furthermore, the last four are the *only* tractable cases of $\text{SAT}_{\text{C}}(S)$.

Theorem 2.2: [Dichotomy Theorems, [Sch78]]
Let S be a finite set of logical relations.

If S is 0-valid or 1-valid or Schaefer, then $\text{SAT}(S)$ is solvable in polynomial time; otherwise, it is NP*-complete.*

If S is Schaefer, then $\text{SAT}_{\text{C}}(S)$ is solvable in polynomial time; otherwise, it is is NP*-complete.*

Theorem 2.2 immediately implies that POSITIVE-1-IN-3-SAT is NP-complete, since this is the same problem as $\text{SAT}(R_{1/3})$, and $R_{1/3}$ is neither 0-valid, nor 1-valid, nor Schaefer, as can be seen by applying the aforementioned closure properties.

To obtain the above dichotomy theorems, Schaefer had to first establish a result asserting that every non-Schaefer set S has extremely high expressive power, in the sense that every logical relation can be defined from an $\mathcal{F}_{\text{C}}(S)$-formula using existential quantification.

Theorem 2.3: [Expressibility Theorem, [Sch78]]
Let S be a finite set of logical relations. If S is non-Schaefer, then for every k-ary logical relation R there is an $\mathcal{F}_{\text{C}}(S)$-formula $\varphi(x_1, \ldots, x_k, z_1, \ldots, z_m)$ such that R coincides with the set of all truth assignments to the variables $x_1, \ldots, x_k$ that satisfy the formula $(\exists \bar{z})\varphi(\bar{x}, \bar{z})$.

3 Propositional Circumscription

In circumscription, properties are specified in some logical formalism, a natural partial order between models of each formula is considered, and the focus is on models that are minimal with respect to this partial order. Minimal models are preferred because they have as few "exceptions" as possible and thus embody common sense. In propositional circumscription, properties are specified using propositional formulas and the focus is on models that are minimal with respect to the coordinate-wise partial order between truth assignments, as defined below.

Let $k \geq 1$ be an integer and let $\alpha = (a_1, \ldots, a_k)$, $\beta = (b_1, \ldots, b_k)$ be two k-tuples in $\{0, 1\}^k$. We write $\beta \leq \alpha$ to denote that, for every $i \leq k$, we have that $b_i \leq a_i$ (as usual, $0 \leq 1$). Also, $\beta < \alpha$ means that $\beta \leq \alpha$ and $\beta \neq \alpha$. If φ is a propositional formula and α is a truth assignment to the variables of φ, then we say that α is a *minimal model* of φ if α satisfies φ and no truth assignment $\beta < \alpha$ satisfies φ.

Let φ and ψ be two propositional formulas in CNF. We say that ψ *can be inferred from* φ *under propositional circumscription*, and write $\varphi \models_{\text{CIRC}} \psi$, if ψ is true in every minimal model of φ. Clearly, if ψ is a conjunction $\bigwedge_{i=1}^{m} c_i$ of clauses c_i, then $\varphi \models_{\text{CIRC}} \psi$ if and only if $\varphi \models_{\text{CIRC}} c_i$, for every $i \leq m$. Thus, the inference problem for propositional circumscription can be stated as follows: given a propositional formula φ in CNF and a clause ψ, does $\varphi \models_{\text{CIRC}} \psi$? Since testing a truth assignment for minimality is in coNP, it follows that the inference problem for propositional circumscription is in Π_2^{P}. As mentioned earlier, in [EG93] this problem was shown to be Π_2^{P}-complete, even when φ is a 3CNF-formula and ψ is just a negative literal $\neg u$. Our goal is to investigate the complexity of the inference problem for propositional circumscription in the context of Schaefer's framework. More precisely, each set S of logical relations gives rise to the following decision problem $\text{INF-CIRC}(\mathcal{F}_{\text{C}}(S))$: given an $\mathcal{F}_{\text{C}}(S)$-formula φ and a clause ψ, does $\varphi \models_{\text{CIRC}} \psi$? The next proposition asserts that each of these decision problems is equivalent to a special case of it.

Proposition 3.1: *For every set S of logical relations,* INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is equivalent to the following decision problem: given an $\mathcal{F}_{\mathrm{C}}(S)$-formula φ and a negative clause $(\neg u_1 \vee \cdots \vee \neg u_n)$, does $\varphi \models_{\mathrm{CIRC}} (\neg u_1 \vee \cdots \vee \neg u_n)$?*

Proof: Given an $\mathcal{F}_{\mathrm{C}}$-formula φ and a clause $(x_1 \vee \cdots \vee x_m \vee \neg u_1 \vee \cdots \vee \neg u_n)$, let φ' be the $\mathcal{F}_{\mathrm{C}}$-formula obtained from φ by replacing each occurrence of x_i, $1 \leq i \leq m$, by **0**. It is easy to verify that $\varphi \models_{\mathrm{CIRC}} (x_1 \vee \cdots \vee x_m \vee \neg u_1 \vee \cdots \vee \neg y_n)$ if and only if $\varphi' \models_{\mathrm{CIRC}} (\neg u_1 \vee \cdots \vee u_n)$. ∎

Consider the following restricted case of INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$): given an $\mathcal{F}_{\mathrm{C}}(S)$-formula φ and a positive clause $(x_1 \vee \cdots \vee x_m)$, does $\varphi \models_{\mathrm{CIRC}} (x_1 \vee \cdots \vee x_m)$? This problem is in coNP, because it is easy to check that $\varphi \models_{\mathrm{CIRC}} (x_1 \vee \cdots \vee x_m)$ if and only if $\varphi \models (x_1 \vee \cdots \vee x_m)$. Thus, the inference of clauses with negative literals is essential in establishing that certain INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) problems are Π_2^{P}-complete.

We are now ready to state the main results of this paper. These results classify the complexity of all INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) problems and, in particular, give efficiently checkable criteria that characterize when INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is a Π_2^{P}-complete problem. As mentioned in Section 1, we first establish a dichotomy theorem for INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$), where S is assumed to be a 1-valid set of logical relations, i.e., every relation in S contains the all-ones tuple $(1, 1, \ldots, 1)$.

Theorem 3.2: *Let S be a 1-valid set of logical relations.*

If S is Schaefer, then INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is in* coNP*; otherwise, it is Π_2^{P}-complete. Actually, if S is non-Schaefer, then even the following special case of* INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is Π_2^{P}-complete: given an $\mathcal{F}_{\mathrm{C}}(S)$-formula φ and a negative literal $\neg u$, does $\varphi \models_{\mathrm{CIRC}} \neg u$?*

Moreover, there is a polynomial-time algorithm to decide whether, given a finite 1-valid set of logical relations, INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is in* coNP *or Π_2^{P}-complete.*

An outline of the proof of Theorem 3.2 is presented in Section 4. The following examples illustrate the preceding Theorem 3.2 and provide new instances of restricted cases of the inference problem for propositional circumscription having the same inherent complexity as the general case.

Example 3.3: Consider the ternary logical relation $K = \{(1,1,1), (0,1,0), (0,0,1)\}$. Using the closure properties that characterize when a logical relation is 2CNF, Horn, dual Horn, or affine, it is easy to see that K is none of the above. For instance, K is not Horn because $(0,1,0) \wedge (0,0,1) = (0,0,0) \notin K$. Consequently, Theorem 3.2 implies that INF-CIRC($\mathcal{F}_{\mathrm{C}}(\{K\})$) is Π_2^{P}-complete. ∎

Example 3.4: Consider the 1-valid set $S = \{R_0, R_1, R_2\}$, where $R_0 = \{0,1\}^3 - \{(0,0,0)\}$ (expressing the clause $(x \vee y \vee z)$), $R_1 = \{0,1\}^3 - \{(1,0,0)\}$ (expressing the clause $(\neg x \vee y \vee z)$), $R_2 = \{0,1\}^3 - \{(1,1,0)\}$ (expressing the clause $(\neg x \vee \neg y \vee z)$). Using the closure properties, it is easy to verify that R_1 is neither 2CNF, nor Horn, nor affine, and that R_2 is not dual Horn. Consequently, Theorem 3.2 implies that INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) is Π_2^{P}-complete. ∎

As mentioned in Section 1, Theorem 3.2 can be used as stepping stone to obtain a dichotomy theorem for the family of all INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) problems, where S is an arbitrary set of logical relations. To this effect, we use the following crucial concept, which was first introduced in [KK01].

Definition 3.5: Let R be a k-ary logical relation. We say that a logical relation T is a *0-section* of R if either T is the relation R itself or T can be defined from the formula $R(x_1, \ldots, x_k)$ by replacing at least one, but not all, of the variables $x_1, \ldots, x_k$ by 0. ∎

To illustrate this concept, consider the logical relation $R_{1/3} = \{(1,0,0), (0,1,0), (0,0,1)\}$. Then the logical relation $\{1\}$ is a 0-section of $R_{1/3}$, since it is definable by $R_{1/3}(x_1, 0, 0)$. In fact, it is easy to see that $\{(1)\}$ is the *only* logical relation that is both 1-valid and a 0-section of $R_{1/3}$.

Theorem 3.6: *Let S be a set of logical relations and let S^* be the set of all logical relations P such that P is both 1-valid and a 0-section of some relation in S.*

If S^* is Schaefer, then INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is in* coNP*; otherwise, it is Π_2^{P}-complete. Actually, if S^* is non-Schaefer, then even the following special case of* INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is Π_2^{P}-complete: given an $\mathcal{F}_{\mathrm{C}}(S)$-formula φ and a negative literal $\neg u$, does $\varphi \models_{\mathrm{CIRC}} \neg u$?*

Moreover, there is a polynomial-time algorithm to decide whether, given a finite set S of logical relations, INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) *is in* coNP *or Π_2^{P}-complete.*

The proof of Theorem 3.6 will be given in the full paper. We now present several different examples that illustrate the power of Theorem 3.6. The first shows how the main result in [EG93] can be easily derived from Theorem 3.6.

Example 3.7: Recall that 3-SAT coincides with SAT(S), where $S = \{R_0, R_1, R_2, R_3\}$ and $R_0 = \{0,1\}^3 - \{(0,0,0)\}$ (expressing the clause $(x \vee y \vee z)$), $R_1 = \{0,1\}^3 - \{(1,0,0)\}$ (expressing the clause $(\neg x \vee y \vee z)$), $R_2 = \{0,1\}^3 - \{(1,1,0)\}$ (expressing the clause $(\neg x \vee \neg y \vee z)$), and $R_3 = \{0,1\}^3 - \{(1,1,1)\}$ (expressing the clause $(\neg x \vee \neg y \vee \neg z)$).

Since the logical relations R_0, R_1, R_2 are 1-valid, they are members of S^*. It follows that S^* is not Schaefer, since R_1 is not 2CNF or Horn or affine, and R_2 is not dual Horn. Theorem 3.6 immediately implies that INF-CIRC($\mathcal{F}_{\mathrm{C}}(S)$) (i.e., INF-CIRC(3CNF)) is Π_2^{P}-complete. ∎

Example 3.8: Consider the set $S = \{R_0, R_3\}$, where R_0 and R_3 are as in the preceding Example 3.7. In this case, SAT(S) is the problem MONOTONE 3-SAT, that is to say, the restriction of 3-SAT to 3CNF-formulas in which every clause is either the disjunction of positive literals or the disjunction of negative literals. It is well known that this problem is NP-complete (this can also be derived from Schaefer's Dichotomy Theorem 2.2). It is not hard to verify that every relation in S^* is dual Horn (for instance, S^* contains R_0, which is dual Horn). Consequently, Theorem 3.6 implies that INF-CIRC($\mathcal{F}_{\subset}(S)$) is in coNP. ∎

The preceding example reveals that the boundary in the dichotomy for the inference problem in classical propositional logic is different than that in the dichotomy for the inference problem in propositional circumscription. Several other instances of this phenomenon are provided by the final example of this section.

Example 3.9: If m and n are two positive integers with $m < n$, then $R_{m/n}$ is the n-ary logical relation consisting of all n-tuples that have m ones and $n - m$ zeros. It is easy to see that $R_{m/n}$ is not Schaefer. Consequently, if S is a set of logical relations each of which is of the form $R_{m/n}$ for some m and n with $m < n$, then SAT(S) is NP-complete. On the other hand, S^* is easily seen to be Horn (and, hence, Schaefer), since every relation P in S^* is a singleton $P = \{(1, \ldots, 1)\}$ consisting of the m-ary all-ones tuple for some m. Consequently, Theorem 3.6 implies that INF-CIRC($\mathcal{F}_{\subset}(S)$) is in coNP.

This family of examples contains POSITIVE-1-IN-3-SAT as the special case where $S = \{R_{1/3}\}$. ∎

4 Outline of Proof of Theorem 3.2

In this section, we present an outline of the dichotomy theorem for INF-CIRC($\mathcal{F}(S)$), where S is a 1-valid set of logical relations. Due to space limitations, we have to confine ourselves to stating the main technical steps and to making a few high-level comments.

Assume first that S is Schaefer. In this case, is easy to see that there is a polynomial-time algorithm to decide whether a given model of an $\mathcal{F}_{\subset}(S)$-formula is minimal. From this fact, it follows immediately that if S is Schaefer, then INF-CIRC($\mathcal{F}_{\subset}(S)$) is in coNP.

Towards the Π_2^{P}-hardness result, assume that S is not Schaefer. Using Schaefer's Expressibility Theorem 2.3, the following decision problem can be shown to be Π_2^{P}-complete: Given a $\mathcal{F}(S)$-formula $\varphi(\bar{x}, \bar{y}, w_0, w_1)$, decide whether the sentence $\forall\bar{x}\exists\bar{y}\varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$ is true. Our goal is to show that this problem has a polynomial-time reduction to INF-CIRC($\mathcal{F}(S)$). One of the key steps in the reduction is the following lemma, which was inspired from a result in [EG93]. A proof can be found in the Appendix.

Lemma 4.1: *Let S be 1-valid set and let $\varphi(\bar{x}, \bar{y}, w_0, w_1)$ be an $\mathcal{F}(S)$-formula, where $\bar{x} = (x_1, \ldots, x_n)$, $\bar{y} = (y_1, \ldots, y_m)$, w_0 and w_1 is the list of its variables. Let u, $\bar{x}' = (x'_1, \ldots, x'_n)$ and $\bar{z} = (z_1, \ldots, z_n)$ be new variables, and let $\chi(u, \bar{x}, \bar{z}, \bar{x}', \bar{y})$ be the following formula*

$$\varphi(\bar{x}', \bar{y}, u/w_0, \mathbf{1}/w_1) \wedge \left(\bigwedge_{i=1}^{n}(x_i \not\equiv z_i)\right) \wedge$$

$$\left(\bigwedge_{j=1}^{m}(u \rightarrow y_j)\right) \wedge \left(\bigwedge_{i=1}^{n}(x'_i \equiv (u \vee x_i))\right).$$

Then the formula $\forall\bar{x}\exists\bar{y}\varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$ is true if and only if $\chi(u, \bar{x}, \bar{z}, \bar{x}', \bar{y}) \models_{\mathrm{CIRC}} \neg u$.

Although φ is an $\mathcal{F}_{\subset}(S)$-formula, the formula χ in the preceding lemma is *not* an $\mathcal{F}_{\subset}(S)$-formula, because it contains elementary connectives, such as $\equiv$, $\rightarrow$, and $\vee$. So, the task now is to construct an $\mathcal{F}_{\subset}(S)$-formula θ in polynomial time such that $\chi \models_{\mathrm{CIRC}} \neg u$ if and only if $\theta \models_{\mathrm{CIRC}} \neg u$. It is now natural to apply Schaefer's Expressibility Theorem 2.3 again and express each of the above elementary connectives using an $\exists\mathcal{F}_{\subset}(S)$-formula, i.e., a formula of the form $\exists\bar{v}\zeta$, where ζ is an $\mathcal{F}_{\subset}(S)$-formula. After these steps are completed, we obtain an $\exists\mathcal{F}_{\subset}(S)$-formula $\exists\bar{v}\chi'$ with the same free variables as χ such that $\chi \models_{\mathrm{CIRC}} \neg u$ if and only if $\exists\bar{v}\chi' \models_{\mathrm{CIRC}} \neg u$. At this point, one may be tempted to simply drop the existential quantifiers $\exists\bar{v}$, focus on the $\exists\mathcal{F}_{\subset}(S)$-formula χ', and claim that $\chi \models_{\mathrm{CIRC}} \neg u$ if and only if $\chi' \models_{\mathrm{CIRC}} \neg u$. The flaw in this argument is that Schaefer's Expressibility Theorem 2.3 gives no explicit information about the possible values of the existential quantifiers in $\exists\mathcal{F}_{\subset}(S)$-formulas expressing logical relations. As a result, the witnesses to the variables $\bar{v}$ in the existential quantifiers $\exists\bar{v}$ may not give rise to minimal satisfying truth assignments of χ', hence the claimed equivalence may fail.

To bypass this serious obstacle, we must give up applying Schaefer's Expressibility Theorem 2.3 and instead have to use certain expressibility lemmas to the effect that all necessary elementary connectives are definable by $\exists\mathcal{F}_{\subset}(S)$-formulas with explicit information about the witnesses to the existential quantifiers. The first of these lemmas, due to Creignou and Hébrard [CH97], concerns the definability of the connectives $\rightarrow$ and $\vee$; it also brings out the importance of the logical relation K introduced in Example 3.3. In what follows, $\mathcal{F}_1(S)$ denotes the class of all formulas obtained from $\mathcal{F}(S)$-formulas by substituting some variables by the constant $\mathbf{1}$.

Lemma 4.2: (Creignou and Hébrard [CH97]) *Let S be a 1-valid, non-Schaefer set of logical relations. Then at least one of the following two statements is true.*

1. *There exists an $\mathcal{F}_1(S)$-formula $\varepsilon(x, y)$ with the property that $(x \rightarrow y) \equiv \varepsilon(x, y)$.*

2. *The logical relation* $K = \{(1,1,1),(0,1,0),(0,0,1)\}$ *is in* $\mathcal{F}_1(S)$*, i.e., there exists an* $\mathcal{F}_1(S)$*-formula* $\kappa(x,y,z)$ *which is satisfied* only *by the three truth assignments* $(1,1,1),(0,1,0)$ *and* $(0,0,1)$*. Therefore:*

   (i) $(x \rightarrow y) \equiv (\exists z)\kappa(x,y,z)$*; moreover,* $(\exists z)\kappa(x,y,z)$ *has the additional property that* 1 *is the only witness for the variable* z *under the truth assignment* $(1,1)$ *to the variables* (x,y)*.*

   (ii) $(x \vee y) \equiv (\exists z)\kappa(z,x,y)$*; moreover,* $(\exists z)\kappa(z,x,y)$ *has the additional property that* 1 *is the only witness for the variable* z *under the truth assignment* $(1,1)$ *to the variables* (x,y)*.*

The second expressibility lemma concerns the definability of the connective $\equiv$.

Lemma 4.3: *Let* S *be a* 1*-valid, non-Schaefer set of logical relations. Then there exists a three-variable* $\mathcal{F}_1(S)$*-formula* $\kappa'(x,y,z)$ *that is satisfied by the truth assignments* $(1,1,1)$*,* $(1,0,0)$ *and* $(0,0,1)$ *but is not satisfied by the truth assignment* $(1,0,1)$ *(no information about the remaining four possible assignments is required). Moreover, if we set* $\lambda(x',u,z,z')$ *to be the formula*

$$(u \rightarrow x') \wedge (x' \vee z) \wedge (z \rightarrow z') \wedge (u \rightarrow z') \wedge \kappa'(x',u,z'),$$

we have the following properties:

(i) the formula $x' \equiv (u \vee \neg z)$ *is logically equivalent to the formula* $(\exists z')\lambda(x',u,z,z')$*;*

(ii) the only witnesses z' *for each of the four assignments* $(x'=1,u=1,z=0),(x'=1,u=0,z=0),(x'=1,u=1,z=1)$ *and* $(x'=0,u=0,z=1)$ *that satisfy the formula* $(\exists z')\lambda(x',u,z,z')$ *are* $z'=1, z'=0, z'=1$ *and* $z'=1$*, respectively.*

The proof of Lemma 4.3 can be found in the Appendix, which also contains a self-contained proof of Lemma 4.2, since that proof is used in the proof of Lemma 4.3.

We are now ready to return to the proof of Theorem 3.2. As stated earlier, our goal is to show that the following problem has a polynomial-time reduction to INF-CIRC$(()\mathcal{F}_C(S))$: given a $\mathcal{F}(S)$-formula $\varphi(\bar{x},\bar{y},w_0,w_1)$, decide whether the sentence $\forall\bar{x}\exists\bar{y}\varphi(\bar{x},\bar{y},\mathbf{0}/w_0,\mathbf{1}/w_1)$ is true. Towards this goal, we start with the formula χ described in Lemma 4.1 and then adjust χ in six successive steps $l = 1,\ldots,6$ (enumerated below). At the last step, we will have constructed an $\mathcal{F}_C(S)$-formula for which the desired reduction holds. More formally, at each step $l = 1,\ldots,6$, we will construct a formula χ_l such that for all $l = 0,\ldots,5$ (assuming that χ_0 is χ), the set of free variables of χ_l is going to be a subset (not necessarily proper) of χ_{l+1} and, in addition, the formulas χ_l will satisfy the following three requirements:

R1: Every truth assignment that satisfies χ_l can be extended to a truth assignment that satisfies χ_{l+1}.

R2: The restriction of every truth assignment that satisfies χ_{l+1} to the variables of χ_l also satisfies χ_l.

R3: Let α and α' be two satisfying truth assignments of χ_l such that $\alpha(u) = 1$ and $\alpha' \leq \alpha$. If β is an extension of α to a satisfying truth assignment of χ_{l+1}, then there is an extension β' of α' to a satisfying truth assignment of χ_{l+1} such that $\beta' \leq \beta$.

It is easy to see that once we prove the above three requirements, then for each $l \geq 0$, χ_l has a minimal satisfying truth assignment with $u = 1$ if and only if χ_{l+1} does. From Lemma 4.1 and the fact that the formula constructed at the last step will be in $\mathcal{F}_C(S)$, it follows that the reduction will be complete.

Notice first that if χ_l and χ_{l+1} have the same set of free variables, then the above three requirements are equivalent to asserting that χ_l and χ_{l+1} are logically equivalent.

Step 1: In χ, replace each connective $x_i' \equiv (u \vee x_i)$, for $i = 1,\ldots,n$, with $x_i' \equiv (u \vee \neg z_i)$. The formula χ_1 has the same variables as χ and it is equivalent to χ, since the conjunct $\bigwedge_{i=1}^{n}(x_i \not\equiv z_i)$ appears in both χ and χ_1 Therefore the requirements R1–R3 are satisfied.

Step 2: In χ_1, replace each connective $x_i' \equiv (u \vee \neg z_i)$, for $i = 1,\ldots,n$, by $\lambda(x_i',u,z_i,z_i')$, where the z_i', for $i = 1,\ldots,n$, are new variables and λ is the formula described in Lemma 4.3. Because of the equivalence of $x_i' \equiv (u \vee \neg z_i)$ with $(\exists z_i')\lambda(x_i',u,z_i,z_i')$, we can immediately conclude that the requirements R1 and R2 are satisfied. To prove requirement R3, observe that because only the variables x_i', u, z_i, for $i = 1,\ldots,n$, are involved in the connectives that are replaced at the current step, and because we have associated a different witness z_i' for each triple of variables x_i', u, z_i, we can restrict our attention to assignments to the three variables x_i', u and z_i only (for an arbitrary but fixed i). Suppose that α and α' are two assignments to x_i', u and z_i such that α' is less than or equal to α and $u = 1$ in α. Then first observe that because of the conjunct $x_i' \equiv (u \vee \neg z_i)$, $x_i' = 1$ in α. Also observe that because of the conjunct $x_i \not\equiv z_i$, the values of z_i in α and α' are equal (recall from the proof of the Key Lemma 4.1 that we express this fact by saying that the value of z_i, as well as x_i, remain "fixed"). The proof of this step can then be completed by distinguishing two cases according to the common value of z_i in α and α'. The details will appear in the full paper.

Step 3: In χ_2, replace each connective $x_i' \vee z_i$ (that appears as part of the formula $\lambda(x_i',u,z_i,z_i')$) by $x_i \rightarrow x_i'$. The satisfaction of the requirments R1–R3 is proved exactly as in Step 1.

Observe that, apart from the conjunct $\bigwedge_{i=1}^{n}(x_i \not\equiv z_i)$, the only logical connectives that have not yet been replaced by an $\mathcal{F}_C(S)$-formula are connectives of the form $x \rightarrow y$

(x and y are used as generic names of variables), where x is either u or x_i or z_i for some i. In the next two steps, we deal with these connectives. Notice first that if the relation $K = \{(1,1,1),(0,1,0,(0,01)\}$ is not in $\mathcal{F}_1(S)$, then we are in Case 1 of Lemma 4.2, therefore there is an $\mathcal{F}_1(S)$ formula $\varepsilon(x,y)$ equivalent to $x \to y$. In this case, in one step that subsumes the following two steps, we just replace every occurrence of $x \to y$ with $\varepsilon(x,y)$. So in the next two steps, we assume that the relation K is in $\mathcal{F}_1(S)$, and therefore we are in Case 2 of Lemma 4.2.

Step 4: In χ_3, replace each connective $u \to x$ (x is again a generic name for variables) with $\kappa(u,x,x')$, where x' is a new variable distinct for each x and κ is the formula described in Case 2 of Lemma 4.2. The validity of the requirements R1 and R2 is immediate. As for requirement R3, restrict attention to the variables u and x, for an arbitrary but fixed variable x. The validity of R3 then follows from the witness property (i) established in Lemma 4.2.

Step 5: Notice first that we cannot imitate Step 4 and replace the connectives of the form $x_i \to x$ with $\kappa(x_i,x,x')$, since in two models α and α' of $x_i \to x$ such that α' is less than or equal to α, the value of x_i remains fixed, while it is the value of x that may drop from 1 in α to 0 in α'. Therefore, the witness property (i) of Lemma 4.2 does not suffice to prove R3 for the case when $x_i = 0$. Instead, we first substitute $x_i \to x$ with $z_i \vee x$ and then substitute the latter with $\kappa(x', z_i, x)$. If we use the witness property (ii) in Lemma 4.2 for the connective $z_i \vee x$, everything goes through, for both possibilities $z_i = 1$ and $z_i = 0$, as it can be easily seen. We deal similarly with the connectives of the form $z_i \to x$.

Step 6: By Schaefer's Expressibility Theorem 2.3, there is an $\mathcal{F}_1(S)$ formula, say $\zeta(x,y,t_1,\ldots,t_s,w_0)$, such that for each $i = 1,\ldots,n$, the connective $x_i \not\equiv z_i$ is logically equivalent to $(\exists \bar{t})\zeta(x_i/x, z_i/y, \bar{t}, 0/w_0)$. To construct χ_6, replace in χ_5 the connectives $x_i \not\equiv z_i$ with $\zeta(x_i/x, z_i/y, x''_{i,1}/t_1,\ldots,x''_{i,s}/t_s, 0/w_0)$, where $x''_{i,r}$ for $i = 1,\ldots,n$ and $r = 1,\ldots,s$ are new variables. It is not hard to see that requirements R1–R3 can be proved in this case with no special properties for the witnesses. Notice that χ_6 is in $\mathcal{F}_C(S)$ (and that the constant 0 was only used in the last step).

This concludes the outline of the proof of Theorem 3.2. ∎

Acknowledgments: We are grateful to Georg Gottlob for bringing to our attention his work with Thomas Eiter on the complexity of circumscription [EG93] and for raising with us the question of a dichotomy in the inference problem for propositional circumscription. We also wish to thank Moshe Y. Vardi for valuable feedback on an earlier version of this paper.

References

[CH96] N. Creignou and M. Hermann. Complexity of generalized satisfiability counting problems. *Information and Computation*, 125(1):1–12, 1996.

[CH97] N. Creignou and J.-J. Hébrard. On generating all solutions of generalized satisfiability problems. *Theoretical Informatics and Applications*, 31(6):499–511, 1997.

[CL94] M. Cadoli and M. Lenzerini. The complexity of closed world reasoning and circumscription. *Journal of Information and System Sciences*, pages 255–301, 1994.

[Cre95] N. Creignou. A dichotomy theorem for maximum generalized satisfiability problems. *Journal of Computer and System Science*, 51(3):511–522, 1995.

[DP92] R. Dechter and J. Pearl. Structure identification in relational data. *Artificial Intelligence*, 48:237–270, 1992.

[EG93] Th. Eiter and G. Gottlob. Propositional circumscription and extended closed-world reasoning are Π_2^P-complete. *Theoretical Computer Science*, 114:231–245, 1993.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[GPP89] M. Gelfond, H. Przymusinska, and T. Przymusinksi. On the relationship between circumscription and negation as failure. *Artificial Intelligence*, 38:273–287, 1989.

[KK01] L.M. Kirousis and Ph.G. Kolaitis. The complexity of minimal satisfiability problems. In *Proc. of the 18th Annual Symposium on Theoretical Aspects of Computer Science - STACS 2001*, volume 2010 of *LNCS*, Springer 2001.

[KS98] D. Kavvadias and M. Sideri. The inverse satisfiability problem. *SIAM Journal of Computing*, 28(1):152–163, 1998.

[KSW97] S. Khanna, M. Sudan, and D.P. Williamson. A complete classification of the approximability of maximization problems derived from Boolean constraint satisfaction. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, pages 11–20, 1997.

[Lad75] R. E. Ladner. On the structure of polynomial time reducibility. *Journal of the Association for Computing Machinery*, 22(1):155–171, 1975.

[McC80] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[RV00] S. Reith and H. Vollmer. Oprimal satisfiability of propositional calculi and constraint satisfaction problems. In *25th Int. Symp. on Mathematical Foundations of Computer Science - MFCS 2000*, volume 1893 of *LNCS*, pages 640–649. Springer, 2000.

[Sch78] T.J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 216–226, 1978.

Appendix: Proof of Lemmas 4.1, 4.2, and 4.3

Lemma 4.1: Let S be 1-valid set and let $\varphi(\bar{x}, \bar{y}, w_0, w_1)$ be an $\mathcal{F}(S)$-formula, where $\bar{x} = (x_1, \ldots, x_n)$, $\bar{y} = (y_1, \ldots, y_m)$, w_0 and w_1 is the list of its variables. Let u, $\bar{x}' = (x'_1, \ldots, x'_n)$ and $\bar{z} = (z_1, \ldots, z_n)$ be new variables, and let $\chi(u, \bar{x}, \bar{z}, \bar{x}', \bar{y})$ be the following formula

$$\varphi(\bar{x}', \bar{y}, u/w_0, \mathbf{1}/w_1) \wedge \left(\bigwedge_{i=1}^{n} (x_i \not\equiv z_i) \right) \wedge$$

$$\left(\bigwedge_{j=1}^{m} (u \rightarrow y_j) \right) \wedge \left(\bigwedge_{i=1}^{n} (x'_i \equiv (u \vee x_i)) \right).$$

Then the formula $\forall \bar{x} \exists \bar{y} \varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$ is true if and only if $\chi(u, \bar{x}, \bar{z}, \bar{x}', \bar{y}) \models_{\mathrm{CIRC}} \neg u$.

Proof: For the if part, consider an assignment α to the variables $\bar{x}$ that satisfies the formula $\forall \bar{y} \neg \varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$. Extend α to an assignment β of all variables of the formula χ by letting $u = 1$, $x'_i = 1$ for $i = 1, \ldots, n$, $y_j = 1$ for $j = 1, \ldots, m$, and by giving to each z_i, for $i = 1, \ldots, n$, the opposite value of x_i. Because φ is 1-valid, it is easy to see that β satisfies χ. We will show that β is actually a minimal satisfying assignment of χ. First observe that the conjuncts $\bigwedge_{i=1}^{n} (x_i \not\equiv z_i)$ ensure that none of the variables $\bar{x}$ or $\bar{z}$ can get a different value at a satisfying assignment of χ strictly smaller than β (we express this fact by saying that the values of $\bar{x}$ and $\bar{z}$ are fixed). Also, the conjuncts $\bigwedge_{j=1}^{m} (u \rightarrow y_j)$ and $\bigwedge_{i=1}^{n} (x'_i \equiv (u \vee x_i))$ ensure that the values of $\bar{y}$ and $\bar{x}'$ are bound to be 1 at any assignment satisfying χ and with $u = 1$. All we have to prove is that u cannot get the value 0 at a satisfying assignment of χ smaller than β. Assume it did and let $\gamma < \beta$ be be a satisfying assignment of χ with $u = 0$. Then, observe that in γ, because of the conjunct $\bigwedge_{i=1}^{n} (x'_i \equiv (u \vee x_i))$, the values of $\bar{x}'$ would be equal to the corresponding values of $\bar{x}$. Therefore, because of the first conjunct of χ, and because $u = 0$ in γ, the values of $\bar{x}$ and $\bar{y}$ in γ would satisfy $\varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$. Now observe that γ and β coincide on $\bar{x}$, because the value of $\bar{x}$ is "fixed". Therefore γ and α also coincide on $\bar{x}$, since by construction β extends α. This is a contradiction, because we assumed that α satisfies $\forall \bar{y} \neg \varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$.

To prove the converse, consider a minimal assignment α of χ with $u = 1$ and also consider the assignment β induced by α on $\bar{x}$. We claim that β satisfies $\forall \bar{y} \neg \varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$. If not, then there is an assignment of values to $\bar{y}$ which combined with β forms an assignment γ that satisfies $\varphi(\bar{x}, \bar{y}, \mathbf{0}/w_0, \mathbf{1}/w_1)$. Extend γ to an assignment δ of all variables of χ by setting $u = 0$, $x'_i = x_i$ for $i = 1, \ldots, n$, and by giving to each z_i for $i = 1, \ldots, n$ the opposite value of x_i. It is easy to see that δ satisfies χ and is strictly smaller than α, which is a contradiction. ∎

Lemma 4.2: (Creignou and Hébrard [CH97]) *Let S be a 1-valid, non-Schaefer set of logical relations. Then at least one of the following two statements is true.*

1. *There exists an $\mathcal{F}_1(S)$-formula $\varepsilon(x, y)$ with the property that $(x \rightarrow y) \equiv \varepsilon(x, y)$.*
2. *The logical relation $K = \{(1,1,1), (0,1,0), (0,0,1)\}$ is in $\mathcal{F}_1(S)$, i.e., there exists an $\mathcal{F}_1(S)$-formula $\kappa(x, y, z)$ which is satisfied* only *by the three truth assignments $(1,1,1)$, $(0,1,0)$ and $(0,0,1)$. Therefore:*

   (i) $(x \rightarrow y) \equiv (\exists z)\kappa(x, y, z)$; moreover, $(\exists z)\kappa(x, y, z)$ has the additional property that 1 is the only witness for the variable z under the truth assignment $(1,1)$ to the variables (x, y).

   (ii) $(x \vee y) \equiv (\exists z)\kappa(z, x, y)$; moreover, $(\exists z)\kappa(z, x, y)$ has the additional property that 1 is the only witness for the variable z under the truth assignment $(1,1)$ to the variables (x, y).

Proof: Since S is a 1-valid, non-Schaefer set of logical relations, it must contain a 1-valid logical relation R that is not affine. As shown in [CH96], there must exist two k-tuples $s, t \in R$ such that $\bar{1} \oplus s \oplus t \notin R$, where $\bar{1}$ is the all-ones k-tuple $(1, \ldots, 1)$ and k is the arity of R. Let $x_1, \ldots, x_k$ be propositional variables and let R' be a relation symbol of arity k that will be interpreted by R. For $(i, j) \in \{0,1\}^2$, let V_{ij} be the set of all variables x_p, $1 \leq p \leq k$, such that the p-th coordinate of the tuple s is equal to i, and the p-th coordinate of the tuple t is equal to j. Let x, y, z, w be four new propositional variables and let $\varphi_1(x, y, z, w)$

be the $\mathcal{F}(S)$-formula $R'(x/V_{00}, y/V_{10}, z/V_{01}, w/V_{11})$ obtained from the formula $R'(x_1, \ldots, x_k)$ by substituting the variable x for all occurrences of the variables in V_{00}, and similarly for the variables y, z, and w. Also let $\varphi_2(x, y, z)$ be the $\mathcal{F}_1(S)$-formula $\varphi_1(x, y, z, \mathbf{1}/w)$. Now observe the following: (1) the truth assignment $(1, 1, 1, 1)$ satisfies $\varphi_1(x, y, z, w)$, since $\bar{1} \in R$; (2) the truth assignment $(0, 1, 0, 1)$ satisfies $\varphi_1(x, y, z, w)$, since $s \in R$; (3) the truth assignment $(0, 0, 1, 1)$ satisfies the $\varphi_1(x, y, z, w)$, since $t \in R$; (4) the truth assignment $(1, 0, 0, 1)$ does not satisfy $\varphi_1(x, y, z, w)$, since $\bar{1} \oplus s \oplus t \notin R$. Therefore, $(1, 1, 1)$, $(0, 1, 0)$ and $(0, 0, 1)$ satisfy $\varphi_2(x, y, z)$, while $(1, 0, 0)$ does not.

We have no information as to whether or not the remaining four assignments $(1, 1, 0), (0, 1, 1), (1, 0, 1)$, $(0, 0, 0)$ satisfy $\varphi_2(x, y, z)$. Thus, we have sixteen possibilities to examine regarding the satisfiability of $\varphi_2(x, y, z)$ by these four truth assignments. We start by branching on the two possibilities for the truth assignment $(0, 0, 0)$:

Case A: $(0, 0, 0)$ satisfies $\varphi_2(x, y, z)$. We distinguish two subcases: Subcase A.1: $(0, 1, 1)$ satisfies $\varphi_2(x, y, z)$. Then set $\varepsilon(x, y) \equiv \varphi_2(x, y, y)$. Subcase A.2: $(0, 1, 1)$ does not satisfy $\varphi_2(x, y, z)$. One more branching: Subcase A.2.1: $(1, 0, 1)$ satisfies $\varphi_2(x, y, z)$.Then set $\varepsilon(x, y) \equiv \varphi_2(y, x, 1)$. Subcase A.2.2: $(1, 0, 1)$ does not satisfy $\varphi_2(x, y, z)$. Then set $\varepsilon(x, y) \equiv \varphi_2(x, y, x)$. This completes the examination of Case A.

Case B: $(0, 0, 0)$ does not satisfy $\varphi_2(x, y, z)$. Consider the following branching: Case B.1: None of the three assignments $(1, 1, 0), (1, 0, 1), (0, 1, 1)$ satisfies $\varphi_2(x, y, z)$. Then $\kappa(x, y, z) \equiv \varphi_2(x, y, z)$. Case B.2: At least one the three assignments $(1, 1, 0), (1, 0, 1), (0, 1, 1)$ satisfies $\varphi_2(x, y, z)$. We make a three-way branching depending on which of these three assignments satisfies $\varphi_2(x, y, z)$. Case B.2.1: $(1, 1, 0)$ satisfies $\varphi_2(x, y, z)$. Then observe that $(x \vee y) \equiv \varphi_2(x, x, y)$. We postpone for a while the continuation of this case where we have already established that $(x \vee y)$ is defined by an $\mathcal{F}_1(S)$-formula. Case B.2.2.: $(1, 0, 1)$ satisfies $\varphi_2(x, y, z)$. Then observe that $(x \vee y) \equiv \varphi_2(x, y, x)$. Again, we postpone the continuation of this case. Case B.2.3: $(0, 1, 1)$ satisfies $\varphi_2(x, y, z)$. Since we have already examined B.2.2, we may assume that $(1, 0, 1)$ does not satisfy $\varphi_2(x, y, z)$. Then set $\varepsilon(x, y) \equiv \varphi_2(x, y, 1)$. At this point all we are left to deal with is the case where $(x \vee y)$ is defined by an $\mathcal{F}_1(S)$-formula. We examine this case below.

Since not every element of S is a dual Horn relation, S must contain a logical relation Q for which there are tuples $s, t \in Q$ such that $s \vee t \notin Q$ (here we use the closure property that characterizes dual Horn relations). By arguments similar to the preceding ones, we can construct an $\mathcal{F}_C(S)$-formula $\psi_2(x, y, z)$ that is satisfied by $(1, 1, 1)$, $(0, 1, 0)$ and $(0, 0, 1)$, but it is not satisfied by $(0, 1, 1)$. Let $\psi_3(x, y, z)$ be the $\mathcal{F}_C(S)$-formula $\psi_2(x, y, z) \wedge (y \vee z)$. Observe that $\psi_3(x, y, z)$ is satisfied by $(1, 1, 1)$, $(0, 1, 0)$ and $(0, 0, 1)$, but it is not satisfied by $(0, 1, 1), (1, 0, 0), (0, 0, 0)$. We are now left with the triples $(1, 1, 0)$ and $(1, 0, 1)$ about which there is no information as to whether they satisfy $\psi_3(x, y, z)$ or not. We consider the following three exhaustive cases:

(1) If $(1, 1, 0)$ satisfies $\psi_3(x, y, z)$, then set $\varepsilon(x, y) \equiv \psi_3(y, 1, x)$; (2) if $(1, 0, 1)$ satisfies $\psi_3(x, y, z)$, then set $\varepsilon(x, y) \equiv \psi_3(y, x, 1)$; (3) if neither $(1, 1, 0)$ nor $(1, 0, 1)$ satisfies $\psi_3(x, y, z)$, then $\kappa(x, y, z) \equiv \psi_3(x, y, z)$. This completes the proof of the Lemma 4.2. ∎

Lemma 4.3: *Let S be a 1-valid, non-Schaefer set of logical relations. Then there exists a three-variable $\mathcal{F}_1(S)$-formula $\kappa'(x, y, z)$ that is satisfied by the truth assignments $(1, 1, 1)$, $(1, 0, 0)$ and $(0, 0, 1)$ but is not satisfied by the truth assignment $(1, 0, 1)$ (no information about the remaining four possible assignments is required). Moreover, if we set $\lambda(x', u, z, z')$ to be the formula*

$$(u \to x') \wedge (x' \vee z) \wedge (z \to z') \wedge (u \to z') \wedge \kappa'(x', u, z'),$$

we have the following properties:

(i) the formula $x' \equiv (u \vee \neg z)$ is logically equivalent to the formula $(\exists z')\lambda(x', u, z, z')$;

(ii) the only witnesses z' for each of the four assignments $(x' = 1, u = 1, z = 0), (x' = 1, u = 0, z = 0), (x' = 1, u = 1, z = 1)$ and $(x' = 0, u = 0, z = 1)$ that satisfy the formula $(\exists z')\lambda(x', u, z, z')$ are $z' = 1, z' = 0, z' = 1$ and $z' = 1$, respectively.

Proof of Lemma 4.3

Let $\kappa'(x, y, z)$ be the formula $\psi_2(y, x, z)$ constructed in the last part of the proof of Lemma 4.2 (notice the inversion of x and y in ψ_2). From the properties of ψ_2, it immediately follows that κ' is satisfied by the truth assignments $(1, 1, 1)$, $(1, 0, 0)$ and $(0, 0, 1)$ but is not satisfied by the truth assignment $(1, 0, 1)$. To prove the properties (i)–(ii), we essentially do exhaustive case analysis for all the possible assignments to the variables x', z, u. We can immediately check that the formula $x' \equiv (u \vee \neg z)$ is satisfied by the assignments $(1, 1, 0), (1, 0, 0), (1, 1, 1)$ and $(0, 0, 1)$ (each bit in each assignment is assigned to x', u and z in this order), while it is not satisfied by the assignments $(0, 1, 0), (0, 0, 0), (0, 1, 1)$ and $(1, 0, 1)$. Now by plugging into the formula $(\exists z')\lambda(x', u, z, z')$ the latter four assignments, one after the other, we can check that they do not satisfy it. In the same way we can check that the former four assignments $(1, 1, 0), (1, 0, 0), (1, 1, 1)$ and $(0, 0, 1)$ do satisfy $(\exists z')\lambda(x', u, z, z')$. During the check that the above four assignments are indeed satisfying, we also determine all possibilities for the witness z', in order to verify that the uniqueness properties required from z' are indeed true (we will only need some of these uniqueness properties). ∎

Relating Semantic and Proof-Theoretic Concepts for Polynomial Time Decidability of Uniform Word Problems

Harald Ganzinger
Max-Planck-Institut für Informatik, Saarbrücken, Germany
hg@mpi-sb.mpg.de

Abstract

In this paper we compare three approaches to polynomial time decidability for uniform word problems for quasi-varieties. Two of the approaches, by Evans and Burris, respectively, are semantical, referring to certain embeddability and axiomatizability properties. The third approach is more proof-theoretic in nature, inspired by McAllester's concept of local inference. We define two closely related notions of locality for equational Horn theories and show that both the criteria by Evans and Burris lie in between these two concepts. In particular, the variant we call stable locality will be shown to subsume both Evans' and Burris' method.

1 Introduction

This paper relates two strands of results about polynomially decidable uniform word problems for quasi-varieties. A quasi-variety is a class of algebras satisfying a particular (in this paper always finite) set $\mathcal{K}$ of equational Horn clauses. Given $\mathcal{K}$, the uniform word problem for $\mathcal{K}$ is to decide whether or not an equational, variable-free Horn clause C, the *query*, is entailed by $\mathcal{K}$: the antecedent of C are the *defining relations* for the *generators* (fresh constants) appearing there; the succedent of C is the word problem to be solved for that *presentation*.

One line of research leading to decidability criteria goes back to work by Skolem (Skolem 1920). Skolem considered the variety of lattices and investigated relational encodings by function-free clauses which we also call Datalog clauses today. Given a Horn theory $\mathcal{K}$ one can flatten the clauses such that all equations in the transformed clauses are of the form $f(x_1,\ldots,x_k)\approx x$ or $x\approx y$ with variables x_i, x, y. Next one can replace functions f by relations (representing their graphs) r^f, so that equations $f(x_1,\ldots,x_k)\approx x$ become atoms $r^f(x_1,\ldots,x_k,x)$. Datalog also allows one to express that equality is an equivalence and that relations are compatible with equality. Moreover, one can specify that function graphs represent partial functions, for example, by saying $r^f(x,y), r^f(x,z) \rightarrow y\approx z$. The "only" property that is lost in the relational encoding is that functions are total. However, if one can show that all *finite* relational models of the encoding can be extended (maintaining $\mathcal{K}$) so that the functions become total, the uniform word problem becomes (polynomially) decidable. For if the relational version C^* of a flat clause C cannot be proved from the Datalog encoding $\mathcal{K}_D$ of $\mathcal{K}$ there will be a finite counter model for $\mathcal{K}_D \cup \neg C^*$ (there are no function symbols other than the constants from C^*), and if that model can be extended to one in which functions are total, this yields a model of $\mathcal{K}$ in which C is false. Skolem presented this technique for the special cases of lattices and for certain axiomatizations of projective geometry, but not for varieties in general. His algorithm for lattices resulting from a dynamic programming implementation of the function-free encoding was rediscovered later by Cosmadakis (1988) and by Freese (1989).[1]

Independently of Skolem's methods, Evans (1951) proved a somewhat stronger result for varieties in general. As Evans' original proof is based on quite different techniques,[2] it is not surprising that Skolem's work is not even mentioned in his paper. Later, Burris (1995) realized that one might, in fact, view Evans' result as a generalization of Skolem's techniques. One of Burris' observations was that a weak form of definedness requirements for the partial functions can also be expressed in Datalog. (For instance, one can require $r^f(x,y) \rightarrow r^g(x,y)$, expressing a relativized definedness property for the function g in terms of the definedness properties of f.) Evans' result is that the uniform word problem is (polynomially) decidable whenever all *finite partial* algebras "satisfying" $\mathcal{K}$ can be injectively embedded into a total $\mathcal{K}$-algebra, where his notion of valid-

[1] This is how Burris (1995) puts it. Looking at the papers, however, the connections to Skolem's work are not so obvious.

[2] Evans' algorithm is ground completion — before the concept of completion was invented by Knuth & Bendix (1970) — of the antecedent of the query together with certain ground instances of the theory clauses dynamically derived from subterms of the query. Using auxiliary constants to name subterms, Evans' procedure is closely related to recent presentations of congruence closure algorithms such as the one by Bachmair & Tiwari (2000).

0-7695-1281-X/01 $10.00 © 2001 IEEE

ity for equations in partial algebras includes precisely those relativized definedness requirements expressible in Datalog (cf. Section 2 below). Having seen the connection between Skolem's and Evans' ideas, it was not difficult for Burris (1995) to extend Evans' result to quasi-varieties. In the same paper he then also presented an even more general criterion for polynomial decidability that refers to the finite axiomatizability of certain classes of substructures of the relational versions of the $\mathcal{K}$-algebras. We will return to this criterion in Section 7 below.

The approaches of Evans and Burris emphasize the rôle of partial algebras (constructed from the subterms and the equations in the antecedent of a query) for the decidability of uniform word problems. An approach that is based on confining deduction to subterms of the query is represented by the concept of local inference systems in (Givan & McAllester 1992, McAllester 1993). Local theories are sets of Horn clauses $\mathcal{K}$ such that $\mathcal{K} \models C$, for variable-free Horn clauses C, only if already $\mathcal{K}_C \models C$, where $\mathcal{K}_C$ is the set of instances of $\mathcal{K}$ in which all terms are subterms of ground terms in either $\mathcal{K}$ or C. Givan and McAllester dealt with non-equational logic whereas we are interested in the equational case. As we shall see below, the main results about non-equational local theories given in (Givan & McAllester 1992, McAllester 1993, Basin & Ganzinger 2001) can be easily extended to the equational case. In particular, the uniform word problem for local equational theories is decidable in polynomial time. A slightly more general variant of this concept is obtained by allowing in local entailment all instances $\mathcal{K}_{[C]}$ of $\mathcal{K}$ by substitutions sending the variables in $\mathcal{K}$-clauses to subterms of the ground terms of C or $\mathcal{K}$. We call $\mathcal{K}$ stably local if already $\mathcal{K}_{[C]} \models C$ whenever $\mathcal{K} \models C$.

The main results of this paper establish close relationships between the approaches by Evans, Burris and McAllester. We show that both Evans' and Burris' criteria lie in between the two variants of locality. The inclusions are (mostly) proper. In particular stable locality is shown to subsume Burris' (and hence Evans') method. We also show for a the subclass of superficial presentations (McAllester 1993) $\mathcal{K}$ that locality and embeddability coincide.

From these results we may conclude that all three criteria for polynomial decidability of uniform word problems are essentially equivalent. In the end, this might not be so surprising given that all three approaches are based on ideas of exploiting the algebraic and deductive structure, respectively, induced by the linearly many query subterms. Moreover it is known that any P-time inference problem can be encoded as a local Horn theory. However, as we shall see below, to clarify the precise relationships induces a number of technical complications mainly related to Evans' specific notion of validity in partial algebras.

2 Basic Notions and Notation

Our investigation assumes an arbitrary, but fixed signature Σ of function symbols to be given, containing an infinite subset C of constants that are used to denote the *generators* in the formulation of word problems. An *equational Horn clause* is an implication of the form $e_1, \ldots, e_k \to e_0$, $k \geq 0$, with equations $e_i = (s_i \approx t_i)$ over Σ. We consider the object language symbol "$\approx$" for formal equality also syntactically as symmetric, so that $s \approx t$ at the same time also denotes $t \approx s$. Sometimes we also take a relational view of functions. Then, given a signature Σ, by Σ^* we denote the corresponding *relational signature* where each n-ary function symbol f in Σ is replaced by a $n+1$-ary relation symbol r^f. If C is an equational Horn clause with all equations of the form $f(x_1, \ldots, x_k) \approx x$ or $x \approx y$, with variables x_i, x, y, by C^* we denote its *relational form*, the Σ^* clause resulting from C by replacing any equation of the form $f(x_1, \ldots, x_k) \approx x$ by an atom $r^f(x_1, \ldots, x_k, x)$. (Equations between variables remain unchanged.)

Let $\mathcal{K}$ be a finite set of clauses, called the *theory*. For technical simplicity we assume that the only terms in $\mathcal{K}$ which are ground are constants. In equational logic this restriction can always be satisfied by flattening transformations, cf. section 4. The *uniform word problem* for $\mathcal{K}$ is to decide if $\mathcal{K} \models C$, for ground Horn clauses C (called *queries*), where "$\models$" denotes implication in first-order logic with equality.

A *partial (Σ)-algebra* is a structure $(A, \{f_A\}_{f \in \Sigma})$, where A is a non-empty set, and for every $f \in \Sigma$ with arity n, f_A is a partial function from A^n to A. [3] Where no confusion about the interpretation of the function symbols can arise, we identify the algebra with its carrier A. For partial algebras the notion of evaluating a term t with respect to a variable assignment β for its variables, yielding a value $\beta(t)$ in A, is the same as for total algebras, except that this evaluation is *undefined*, if $t = f(t_1, \ldots, t_n)$ and either one of the $\beta(t_i)$ is undefined, or else $(\beta(t_1), \ldots, \beta(t_n))$ is not in the domain of f_A. If the term t is ground, the evaluation is independent of any variable assignment, and its value will be denoted by t_A. If $A \subseteq B$ are partial Σ-algebras, B is called an *expansion* of A if $f_A = f_B|_A$, the restriction of the partial function f_B to the subset A. A is called a *(total) algebra* whenever all functions are total. Under the relational view, if A is a (partial or total) Σ-algebra, by A^* we denote its relational variant, the Σ^*-structure for which $r^f_{A^*}(a_1, \ldots, a_n, a)$ is true if, and only if, $f_A(a_1, \ldots, a_n) = a$.

Given a set $\mathcal{K}$ of equational Horn clauses, by $\mathcal{K}$ we also denote the *quasi-variety* represented by $\mathcal{K}$, that is, the class of all total algebras that satisfy (in the usual sense of first-order logic with equality) the clauses in $\mathcal{K}$. A *partial $\mathcal{K}$-algebra* A is a partial algebra satisfying all the clauses in

[3] This also includes the possibility for a constant symbol to not be defined in A.

$\mathcal{K}$. Hereby a clause $s_1 \approx t_1, \ldots, s_k \approx t_k \rightarrow s \approx t$ *is satisfied* (is *valid*) in A, if for all assignments β of elements in A to the variables in the clause, whenever the $\beta(s_i)$ and $\beta(t_i)$ are all defined and $\beta(s_i) = \beta(t_i)$, then

(i) if $\beta(s)$ and $\beta(t)$ are both defined then $\beta(s) = \beta(t)$; and

(ii) if $s = f(u_1, \ldots, u_n)$, $n \geq 0$, and if all terms $\beta(u_i)$ and $\beta(t)$ are defined, then $\beta(s)$ is also defined.[4]

We say that a partial algebra *weakly satisfies* $\mathcal{K}$, if only requirement (i) is satisfied for any clause in $\mathcal{K}$. In a partial $\mathcal{K}$-algebra, requiring that an equation be satisfied also induces certain definedness requirements for the functions that appear in the equation. Sometimes we speak of *strong satisfaction* when we want to emphasize that both (i) and the *definedness requirements* (ii) are fulfilled.

This specific concept of validity for clauses in partial algebras was introduced by Evans. Its definedness requirements may appear ad hoc at first sight. Viewed relationally, however, one observes that this is the strongest notion of relative definedness that can directly be expressed in Datalog. For instance an equation $f(g(x)) \approx h(x)$ can be encoded by writing the two clauses $r^g(x,y), r^f(y,z) \rightarrow r^h(x,z)$ and $r^g(x,y), r^h(x,z) \rightarrow r^f(y,z)$, where these two clauses imply both the equality and the definedness requirement associated with the equation. In other words, the natural encoding of conditional equations into Datalog induces the relativized definedness requirements in Evans' definition.

As an aside, many more notions of validity have been considered in the literature, usually motivated by a particular application. One of the more prominent choices is to consider existential equality, where an equation $s \approx t$ is interpreted as "s and t are defined and are equal". Existential equality appears to be useful for applications to the semantics of programming languages and to intuitionistic logic (Scott 1979). The treatment of partial algebras by Burmeister (1986) is also based on existential equality since most of the other notions of validity can be encoded in existential equality.

A (total) mapping $h : A \rightarrow B$ between partial Σ-algebras A and B is called a *(weak) (Σ-) homomorphism* if whenever $f_A(a_1, \ldots, a_k)$ is defined, then so is $f_B(h(a_1), \ldots, h(a_k))$, and $h(f_A(a_1, \ldots, a_k)) = f_B(h(a_1), \ldots, h(a_k))$. A partial Σ-algebra A is said to *weakly embed* into $\mathcal{K}$ if there exists a (total) $\mathcal{K}$-algebra B and an injective (weak) homomorphism from A to B.

Evans' result (which was later extended to quasi-varieties by Burris) refers to partial algebras with definedness requirements:

THEOREM 2.1 (EVANS 1951, BURRIS 1995) Let $\mathcal{K}$ be a finite set of Horn clauses. If every finite partial $\mathcal{K}$-algebra weakly embeds into $\mathcal{K}$, then the uniform word problem for $\mathcal{K}$ is decidable in polynomial time.

A proof of this theorem, via the relational encoding, was outlined in the introduction.

3 Local Equational Theories

Let Ψ be a set of ground terms and C a clause. By $\mathcal{K}_\Psi$ we denote the set of ground instances of $\mathcal{K}$ in which all terms are in Ψ. We say that $\mathcal{K}$ *entails* C *with respect to* Ψ, and write $\mathcal{K} \models_\Psi C$, if $\mathcal{K}_\Psi \models C$.

If S is a clause or a set of clauses, by $\mathsf{st}[S]$ we denote the set of all ground (sub)terms appearing in S or in $\mathcal{K}$. (We use this notation when $\mathcal{K}$ is fixed by the context. Note that we have restricted theory presentations $\mathcal{K}$ to only contain constants as ground terms.) A theory $\mathcal{K}$ is called *local* if for every ground Horn clause C we have $\mathcal{K} \models C$ if, and only if, $\mathcal{K}_{\mathsf{st}[C]} \models C$. Whenever $\mathcal{K}_{\mathsf{st}[C]} \models C$ we say that C is *locally entailed* by $\mathcal{K}$. The following presentation Int of integers with successor and predecessor is local (at the end of this section we will briefly explain why):

$$\begin{aligned} p(x) \approx y &\rightarrow s(y) \approx x \\ s(x) \approx y &\rightarrow p(y) \approx x \\ p(x) \approx p(y) &\rightarrow x \approx y \\ s(x) \approx s(y) &\rightarrow x \approx y \end{aligned}$$

For a local theory to decide a word problem represented by C it suffices to generate all ground instances of the theory $\mathcal{K}$ in which all terms are either subterms of C or constants in $\mathcal{K}$ and to check whether C is entailed by those ground instances. For example, the query $p(s(z)) \approx z$ is entailed in equational logic by the instance $s(z) \approx s(z) \rightarrow p(s(z)) \approx z$ of the second clause in Int.[5] In that clause, all terms are subterms of the query. The third and fourth clauses of Int are consequences of the first two clauses. For example, $s(u) \approx s(v) \rightarrow u \approx v$ follows from $s(u) \approx s(u) \rightarrow p(s(u)) \approx u$ and $s(v) \approx s(v) \rightarrow p(s(v)) \approx v$. However, in this derivation there appear terms (such as $p(s(u))$) which are not admitted in local entailment. Hence, although the injectivity clauses are entailed by the other clauses, for the presentation to be local they cannot be deleted. This is a general phenomenon. For a presentation to be local, sufficiently many consequences must be present — in particular those consequences which are not entailed by local implication. Clearly, locality is a property of a presentation rather than a property of the quasi-variety.

If the size of $\mathcal{K}$ is considered as a constant, the set $\mathcal{K}_{\mathsf{st}[C]}$ is a finite set of equational ground clauses the size of which is polynomially bounded by C. In the non-equational case

[4] Remember that symmetry of $\approx$ is built into the notation so that the same property is also assumed to hold when exchanging s and t.

[5] Note that z is formally a constant here. But since it does not occur anywhere else, proving $p(s(z)) \approx z$ is the same as showing $\mathsf{Int} \models \forall z\,(p(s(z)) \approx z)$.

when $\approx$ is interpreted as an arbitrary binary relation symbol, applying the result of Dowling & Gallier (1984), we observe that entailment of queries for local theories is decidable in polynomial time. We will show that this result can be extended also to the equational case as local implication is independent of whether or not equality is internal or external.

Let us use $\models$ and $\models_{\mathsf{neq}}$ to denote implication in logic with equality and without equality, respectively. In logic without equality, $\approx$ is an arbitrary binary relation symbol. Let EQ denote the set of equality axioms consisting of reflexivity, symmetry, transitivity and *congruence axioms*

$$x_1 \approx y_1, \ldots, x_k \approx y_k \quad \rightarrow \quad f(x_1, \ldots, x_k) \approx f(y_1, \ldots, y_k)$$

for each k-ary function symbol f in the signature. In first-order logic equality can be internalized since $\mathcal{K} \models C$ if, and only if, $\mathcal{K} \cup EQ \models_{\mathsf{neq}} C$. This carries over to local implication, the main reason being that EQ itself is a local theory (in logic without equality):

PROPOSITION 3.1 (GIVAN & MCALLESTER 1992) For any ground Horn clause C we have $EQ \models_{\mathsf{neq}} C$ if, and only if, $EQ_{\mathsf{st}[C]} \models_{\mathsf{neq}} C$.

A consequence of this result is that congruence closure, that is, the uniform word problem for the class of all Σ-algebras, is decidable in polynomial time, a result that was first proved by Kozen (1977) and later shown to be in $O(n \log n)$ by Downey, Sethi & Tarjan (1980).

PROPOSITION 3.2 Let S be a set of Horn clauses in which all terms are contained in a subterm-closed set Ψ of ground terms. For equalities e between terms in Ψ we have $S \models e$ if, and only if, $S \cup EQ_\Psi \models_{\mathsf{neq}} e$.

Proof. The direction from right to left is trivial. Conversely, suppose that $S \models e$ in equational logic. Then $\bigcup_i \mathcal{T}_S^i(\emptyset) \cup EQ \models_{\mathsf{neq}} e$, with $\mathcal{T}_S$ the immediate consequence operator sending interpretations I to

$$\{e_0 \mid I \cup EQ \models_{\mathsf{neq}} e_i, \text{ for some clause } e_1, \ldots, e_k \rightarrow e_0 \text{ in } S\}.$$

From Proposition 3.1 we infer that $I \cup EQ \models_{\mathsf{neq}} e_i$ only if $I \cup EQ_{\Psi_i} \models_{\mathsf{neq}} e_i$, where Ψ_i is the set of all subterms in I or in e_i. These terms are all in Ψ for those I obtained as $\mathcal{T}_S^n(\emptyset)$, as an easy induction shows. Therefore, $S \cup EQ_\Psi \models_{\mathsf{neq}} e$. □

As an immediate consequence we obtain:

THEOREM 3.3 Let S be a set of Horn clauses. Then S is a local theory in logic with equality if, and only if, $S \cup EQ$ is local in logic without equality.

This property of equational logic allows us to extend the results by Givan & McAllester (1992), McAllester (1993) and Basin & Ganzinger (2001) to local equational theories: Any language in P can be encoded as a uniform word problem for a local theory, that is, the method is complete for polynomial time. The set of local equational Horn theories is co-recursively enumerable but undecidable (McAllester 1993). Recursively enumerable approximations of the class of local theories as given in (McAllester 1993, Basin & Ganzinger 2001) can be easily adapted to the equational case. In particular we may use the Saturate system (Ganzinger, Nieuwenhuis & Nivela 1994) to saturate non-local presentations as described in (Basin & Ganzinger 2001). The locality of the Int example was demonstrated by Saturate by checking that all ordered resolution inferences between the clauses in $\mathsf{Int} \cup EQ$ are redundant in that the respective consequences of $\mathsf{Int} \cup EQ$ are entailed by smaller instances of $\mathsf{Int} \cup EQ$. This was checked for all total and well-founded extensions of the subterm ordering so that by the criterion given in (Basin & Ganzinger 2001) the locality of $\mathsf{Int} \cup EQ$ follows.

Queries C for local equational theories $\mathcal{K}$ are decidable in polynomial time by applying dynamic programming à la Dowling & Gallier (1984) to the clauses in $(S \cup EQ)_{\mathsf{st}[C]}$. Note however that this implementation method will always give at least cubic complexity as $|EQ_{\mathsf{st}[C]}|$ is in $\Omega(n^3)$ if n is the number of terms in C. For practical applications, in particular to problems arising in program analysis (McAllester 1999), more efficient equational reasoning is required. Recent results into this direction, extending the congruence closure method of Downey et al. (1980) to conditional equations, are given in (Ganzinger & McAllester 2001).

4 Flattening and Linearity

A quasi-variety $\mathcal{K}$ is local if queries C are implied already by those ground instances of $\mathcal{K}$ in which all terms are subterms of C or $\mathcal{K}$. In the equational case this property, if it is true, has to be invariant under transformations of C modulo equality. In particular, *flattening transformations* of C, replacing $C[f(\ldots, t, \ldots)]$ by $C' = c \not\approx t \vee C[f(\ldots, c, \ldots)]$, where c is a fresh constant, do not affect entailment from $\mathcal{K}$, but will change the set Ψ of terms allowed in a local proof.

A ground clause is called *flat* if its terms have depth at most 2. A flat ground clause is called *linear* if whenever a constant occurs in two functional terms in the clause, the two terms are identical, and if no term contains two occurrences of a constant. Hence the clause $c \approx f(a,b) \rightarrow f(a,b) \approx f(b,a)$ is flat but not linear. If the clause occurs as a query, an equivalent linear query would be $a \approx a', b \approx b', c \approx f(a,b) \rightarrow f(a,b) \approx f(b',a')$, where a' and b' are fresh constants. For theory clauses the definition is essentially the same, with variables playing the rôle of

constants: We say that a theory clause in $\mathcal{K}$ is *flat*, whenever function symbols (including constants) only occur as arguments of the equality symbol, but not as arguments of function symbols. A flat theory clause is called *linear* if whenever a variable occurs in two functional terms, the two terms are identical, and if no term contains two occurrences of a variable. Hence $f(x,y) \approx f(x,a)$ is neither flat nor linear. An equivalent flat and linear clause is $x' \approx x, z \approx a \rightarrow f(x,y) \approx f(x',z)$, where x' and z are fresh variables. Clearly all clauses, queries as well as theory clauses, can be flattened (and linearized) by the introduction of auxiliary constants and variables, respectively. If $\mathcal{K}$ is a Horn theory, by $\mathcal{K}_{\mathsf{flin}}$ we denote the set of flat and linear instances (not necessarily ground) of the clauses in $\mathcal{K}$. Clearly, a non-flat clause cannot have any flat instances. A flat but non-linear clause such as $a \approx f(x,y) \rightarrow b \approx f(x',y)$ has the flat and linear instance $a \approx f(x,y) \rightarrow b \approx f(x,y)$. Therefore, if $\mathcal{K}$ is finite and if subsumed clauses are ignored, $\mathcal{K}_{\mathsf{flin}}$ is also finite.

PROPOSITION 4.1 (i) If $\mathcal{K}$ is a local theory then $\mathcal{K}_{\mathsf{flin}}$ is also local. In this case, for any query C, it holds that $\mathcal{K} \models C$ if, and only if, $\mathcal{K}_{\mathsf{flin}} \models C$.
(ii) If $\mathcal{K}$ locally entails any flat and linear query C that is entailed by $\mathcal{K}$, then $\mathcal{K}$ is local.

Proof. (i) Suppose that $\mathcal{K}$ is local. If $\mathcal{K} \models C$ then $\mathcal{K} \models \mathsf{flin}(C)$, where $\mathsf{flin}(C)$ is the result of flattening and linearizing C. Since $\mathcal{K}$ is local, we obtain $\mathcal{K}_\Psi \models \mathsf{flin}(C)$, with $\Psi = \mathsf{st}[\mathsf{flin}(C)]$ the set of ground subterms in $\mathsf{flin}(C)$ and $\mathcal{K}$. As all terms in Ψ are flat and linear, and no constant occurs in more than one functional term, the clauses in $\mathcal{K}_\Psi$ are flat and linear. Therefore $\mathcal{K}_\Psi \subseteq (\mathcal{K}_{\mathsf{flin}})_\Psi$, hence $\mathcal{K}_\Psi = (\mathcal{K}_{\mathsf{flin}})_\Psi$. Consequently $(\mathcal{K}_{\mathsf{flin}})_\Psi \models C$ and $\mathcal{K}_{\mathsf{flin}}$ is a local theory.

(ii) Suppose that $\mathcal{K} \models C$. We show that $\mathcal{K}_{\mathsf{st}[C]} \models C$. We may flatten and linearize C into C' by using auxiliary, pairwise different constants c_t not occurring in $\mathcal{K}$ or C, to denote the subterms t of C. Specifically, we may assume that for any original subterm $t = f(t_1,\ldots,t_n)$ in C, C' contains the negative equation $c_{f(t_1,\ldots,t_n)} \approx f(c_{t_1},\ldots,c_{t_n})$ defining the constant as an abbreviation for the respective term, and that, apart from these definitions, no other equation in C' contains a functional term. Since $\mathcal{K} \models C'$, by assumption we also have $\mathcal{K}_{\Psi'} \models C'$, where Ψ' is the set of ground terms in C' or $\mathcal{K}$. The only terms that may occur in $\mathcal{K}_{\Psi'}$ are the constants c_t, the constants in $\mathcal{K}$, and terms of the form $f(c_{t_1},\ldots,c_{t_n})$ such that $f(t_1,\ldots,t_n)$ is a subterm in C. Replacing the c_t in $\mathcal{K}_{\Psi'}$ by t, therefore, yields clauses in $\mathcal{K}_{\mathsf{st}[C]}$ which entail C. □

In particular, if $\mathcal{K}$ is local, the quasi-varieties $\mathcal{K}$ and $\mathcal{K}_{\mathsf{flin}}$ coincide as $\mathcal{K}_{\mathsf{flin}}$ also implies those instances of $\mathcal{K}$ which are not in $\mathcal{K}_{\mathsf{flin}}$. (The latter are trivially implied by $\mathcal{K}$.) Part (ii) says that it is sufficient to show local entailment for flat, linear queries in order for a theory to be local. The relevance of this proposition is that when investigating locality for Horn theories it is sufficient to restrict attention to flat and linear theories and queries.

Flattening transformations for theory clauses that transform a clause $C[f(\ldots,t,\ldots)]$ in $\mathcal{K}$ into $C' = x \not\approx t \vee C[f(\ldots,x,\ldots)]$, where x is a fresh variable, neither change the class of total nor the class of partial $\mathcal{K}$-algebras. The same holds for linearization transformations, replacing $C[f(\ldots,y,\ldots)]$, with y a variable, by $C' = x \not\approx y \vee C[f(\ldots,x,\ldots)]$, where x is a fresh variable. However replacing $\Gamma \rightarrow f(\ldots) \approx t$ by $\Gamma, x \approx f(\ldots) \rightarrow x \approx t$, although not affecting the class of total $\mathcal{K}$-algebras, only preserves weak satisfaction in partial algebras. Strong satisfaction which might induce that certain f-terms be defined, are made void when this kind of transformation is performed.

5 Stably Local Theories

The proposition 4.1 also suggests that the definition of locality is sometimes too strong. In fact, the following less restrictive form of locality, where we allow arbitrary query subterms to be instantiated for the variables in theory clauses, will also be useful. Let $\mathcal{K}_{[C]}$, for C a ground clause, denote the set of ground instances of clauses in $\mathcal{K}$ where variables are mapped to terms in $\mathsf{st}[C]$, that is, to subterms in C or constants in $\mathcal{K}$. Considering $\mathcal{K}_{[C]}$, we also have instances of $\mathcal{K}$ at our disposal in which there are terms not in $\mathsf{st}[C]$. For example, if $C = a \approx b$ and if $f(x,y) \approx f(y,x)$ is in $\mathcal{K}$ then $f(a,b) \approx f(b,a)$ is in $\mathcal{K}_{[C]}$ but not in $\mathcal{K}_{\mathsf{st}[C]}$, since $f(a,b)$ is not a term in C. We say that $\mathcal{K}$ is *stably local* if for every ground Horn clause C we have $\mathcal{K} \models C$ if, and only if, $\mathcal{K}_{[C]} \models C$. This presentation Int' of integers with successor and predecessor is stably local even without the presence of the injectivity clauses for s and p:

$$\begin{aligned} p(x) \approx y &\rightarrow s(y) \approx x \\ s(x) \approx y &\rightarrow p(y) \approx x \end{aligned}$$

In fact, $s(u) \approx s(v) \rightarrow u \approx v$, say, follows from $s(u) \approx s(u) \rightarrow p(s(u)) \approx u$ and $s(v) \approx s(v) \rightarrow p(s(v)) \approx v$, where these instances of Int' are admitted in stably local entailment but not in local entailment. Rewriting the clauses of Int' into

$$\begin{aligned} &\rightarrow s(p(x)) \approx x \\ &\rightarrow p(s(x)) \approx x \end{aligned}$$

gives another stably local (non-flat) presentation Int'' of the integers. For example, $p(u) \approx v \rightarrow s(v) \approx u$ is stably locally entailed by the instance $s(p(u)) \approx u$ of the first clause in Int''.

Locality is a special case of stable locality since $\mathcal{K}_{\mathsf{st}[C]} = \mathcal{K}_{[C]_{\mathsf{st}[C]}}$. Stable locality is insensitive towards flattening of goals in that for every theory $\mathcal{K}$ we have $\mathcal{K}_{[C]} \models C$ iff $\mathcal{K}_{[\mathsf{flin}(C)]} \models \mathsf{flin}(C)$. Like locality, stable locality also implies

that the uniform word problem is decidable in polynomial time.

THEOREM 5.1 Let $\mathcal{K}$ be a given theory the size of which is considered constant. If $\mathcal{K}$ is stably local and if C is a ground clause then $\mathcal{K} \models C$ can be decided in time $O(n^{3k})$ where n is the size of C and k the maximal number of variables in any clause in $\mathcal{K}$.

Proof. Let $C = \Gamma \rightarrow e$. By stable locality we have $\mathcal{K} \models C$ if, and only if, $\mathcal{K}_{[C]} \cup \Gamma \models e$. From Proposition 3.2 we infer that the latter is equivalent with $\mathcal{K}_{[C]} \cup \Gamma \cup EQ_{\mathsf{st}[\mathcal{K}_{[C]}] \cup \mathsf{st}[C]} \models_{\mathsf{neq}} e$. As the number of terms appearing $\mathcal{K}_{[C]}$ is in $O(n^k)$, the size of this set of propositional Horn clauses is in $O(n^{3k})$. □

As Int'' is stably local we obtain a cubic upper bound for the uniform word problem for integers with s and p.

Refined complexity bounds can be obtained by more precise analysis of the term structure in $\mathcal{K}$. Although important in practice, this is not our concern here. Also, with a specialized treatment of equality one can get a better complexity bound in many cases. Using congruence closure to directly decide $\mathcal{K}_{[C]} \cup \Gamma \models e$ would yield a much better complexity of $O(n \log n)$ for $\mathcal{K} = \mathsf{Int}'''$.

6 Locality and Weak Embeddability

In this section we establish the main relationships between Evans' embeddability criterion and locality. We will show that Evans' criterion is weaker than stable locality but stronger than locality. For a large subclass of presentations, locality and Evans' criterion coincide. We also show that the weaker form of Evans' criterion with satisfaction replaced by weak satisfaction is equivalent with locality.

Looking at the proofs in (Evans 1951) it is not surprising that some sort of relation exists between embeddability and locality. However the precise details are not so straightforward, the reason being that Evan's notion of validity, involving a semantic notion of definedness, is not so easily captured proof-theoretically. A special case is the definedness of theory constants. In this section we will additionally require that for a partial algebra A in order to satisfy, or weakly satisfy, a theory $\mathcal{K}$, *every constant appearing in $\mathcal{K}$ is defined in A*. With this, Evans' criterion becomes even stronger as fewer partial algebras need to be embedded. The restriction will only be needed for the proof of Theorem 6.1 and its applications in Section 7.

6.1 Locality Implies Embeddability

In the following theorem, under the assumption of locality, the embeddability property is even shown for infinite partial algebras that need only weakly satisfy $\mathcal{K}$.

THEOREM 6.1 Let $\mathcal{K}$ be a local set of flat Horn clauses. Then every partial algebra which weakly satisfies $\mathcal{K}$ weakly embeds into $\mathcal{K}$.

Proof. We prove the contrapositive of the theorem. Let A be a partial algebra weakly satisfying $\mathcal{K}$ that does not weakly embed into $\mathcal{K}$. We will show that then $\mathcal{K}$ is not local. Without loss of generality we may assume that $A \subseteq C$, that is, the elements of A are generators in Σ, but no constant occurring in $\mathcal{K}$ is a member of A. Moreover let Γ_A be the "table" of the function definitions in A, that is, the set of equations of the form $f(a_1, \ldots, a_n) \approx a$ with a, a_j in A and f a function symbol in Σ, such that $f_A(a_1, \ldots, a_n)$ is defined and equal to a. Suppose I is a Σ-algebra satisfying $\mathcal{K}$ and also the equations in Γ_A. The mapping h sending a in A to its value a_I in I is a weak Σ-homomorphism as I satisfies Γ_A. By assumption, A does not weakly embed into I so that there are two different elements a and a' of A for which $I \models a \approx a'$. Hence whatever model of $\mathcal{K} \cup \Gamma_A$ one chooses, it will identify two constants corresponding to different elements in A. In other words, $\mathcal{K} \cup \Gamma_A \models \bigvee_{a \neq a'} a \approx a'$. Since $\mathcal{K} \cup \Gamma_A$ is a Horn theory, one of the disjuncts must be entailed, that is, $\mathcal{K} \cup \Gamma_A \models a \approx a'$, for two different elements a and a' in A. Compactness of first-order logic ensures that only finitely many equations in Γ_A are needed to deduce $a \approx a'$. We have shown that there is a (finite) Horn clause $C = \Gamma \rightarrow a \approx a'$ such that $\mathcal{K} \models C$, and with Γ true, but $a \approx a'$ false in A.

Suppose that already $\mathcal{K}_\Psi \models \Gamma \rightarrow a \approx a'$, with Ψ the set of ground terms in $\mathcal{K}$ or C. By assumption, A weakly satisfies $\mathcal{K}$. Moreover, all the terms occurring in $\mathcal{K}_\Psi$ and Γ are defined in A. Therefore, every equation in defined ground terms that is true in the least congruence generated by $\mathcal{K}_\Psi \cup \Gamma$ is also true in A.[6] But this implies that a and a' are equal in A which is not the case. Consequently, $\mathcal{K}_\Psi \not\models \Gamma \rightarrow a \approx a'$, hence $\mathcal{K}$ is not a local theory. □

Hence locality is subsumed by Evans' criterion. This subsumption relation is proper. For the presentation Int' one can show that every finite partial Int'-algebra weakly embeds into Int'. (In any partial Int'-algebra, s $[p]$ must be defined on all p $[s]$ images. Therefore both partial functions have to be injective.) However, as we have seen before, Int' is only stably local but not local.

In the proof of the above theorem it is crucial that theory constants are defined in partial algebras that weakly satisfy $\mathcal{K}$. Suppose we have $\mathcal{K}$ consisting of the two clauses

$$a \approx a \rightarrow a \approx b$$
$$a \approx b \rightarrow a \approx c\,.$$

Since $\mathcal{K}$ is equivalent to the two ground equations $a \approx b$ and $a \approx c$, $\mathcal{K}$ is a local theory. But if F is a partial algebra in which a is undefined and b and c are defined but different, F vacuously satisfies $\mathcal{K}$ (including definedness requirements), yet cannot be weakly embedded into $\mathcal{K}$.

[6] If a partial algebra A satisfies a set S of ground Horn clauses and if every term in S is defined in A, then if $S \models s \approx t$, with s and t defined in A, then $A \models s \approx t$. As equality is a local theory, cf. Proposition 3.1, equational reasoning can be confined to the subterms in S which are all defined in F.

6.2 Embeddability Implies Locality

THEOREM 6.2 Let $\mathcal{K}$ be a set of flat, linear Horn clauses. Suppose that every finite partial algebra which weakly satisfies $\mathcal{K}$ weakly embeds into $\mathcal{K}$. Then $\mathcal{K}$ is local.

Proof. Using the proposition 4.1, part (ii), we have to show that, under the given assumptions, if $\mathcal{K} \models C$, then $\mathcal{K}_{\mathsf{st}[C]} \models C$, for flat and linear ground clauses C. Let Ψ be shorthand notation for $\mathsf{st}[C]$. As C and the clauses in $\mathcal{K}$ are flat, a term in Ψ is either a constant, or else of the form $f(c_1,\ldots,c_n)$, with constants c_i, $n \geq 0$. Let $C = s_1 \approx t_1,\ldots,s_k \approx t_k \rightarrow s \approx t$, and let us assume, for the purpose of deriving a contradiction, that C is not entailed by $\mathcal{K}_\Psi$. Then there exists an algebra I satisfying $\mathcal{K}_\Psi$ and the equations $s_i \approx t_i$, but s and t are different in I, that is, I satisfies $s \not\approx t$. From this we will now construct a finite, partial algebra F satisfying $s_i \approx t_i$ and $s \not\approx t$ and weakly satisfying $\mathcal{K}$.

Let $F = \{t_I \mid t \text{ a term in } \Psi\}$, and let the functions f in Σ be defined by $f_F(a_1,\ldots,a_n) = f(c_1,\ldots,c_n)_I$, with $n \geq 0$, whenever there exist constants c_i in Ψ such that $a_i = c_{iI}$, for $1 \leq i \leq n$, and $f(c_1,\ldots,c_n)$ is also a term in Ψ. Let f_F be undefined in all other cases. We now show that F weakly satisfies $\mathcal{K}$. (By construction, F satisfies the $s_i \approx t_i$ as well as $s \not\approx t$.) Clearly, the constants appearing in $\mathcal{K}$ are defined in F. Now let $D = u_1 \approx v_1,\ldots,u_m \approx v_m \rightarrow u \approx v$ be a clause in $\mathcal{K}$ and let β be an assignment of elements in F to the variables in D such that the $\beta(u_i) = \beta(v_i)$, with all these terms defined. We can now find a substitution σ of the variables in D by terms in Ψ such that for every term w in D, whenever $\beta(w)$ is defined then $w\sigma$ is a term in Ψ and $(w\sigma)_I = \beta(w)$. For instance, if a w is of the form $f(x_1,\ldots,x_n)$, choose $x_j\sigma$ to be a constant c_j in Ψ such that $c_{jI} = \beta(x_j)$, for $1 \leq j \leq n$, and $f(c_1,\ldots,c_n)$ is also a term in Ψ. (Note that the arguments to f have to be pairwise distinct variables.) By the definition of f_F, such constants can be found whenever f_F is defined on the $\beta(x_i)$. With this, $f(x_1,\ldots,x_n)\sigma$ is in fact a term in Ψ. As the entire clause D is linear in that no variable occurs in two different functional terms, the σ for the individual occurrences of functional terms can be combined into a single substitution. For variables y which do not occur in a functional term in D, the substitution can be an arbitrary term s in Ψ such that $s_I = \beta(y)$.

We have to verify the condition (i) in the definition of satisfaction for clauses in partial algebras. Suppose that $\beta(u)$ and $\beta(v)$ are both defined. By the construction of σ we have that $u\sigma$ and $v\sigma$ are in Ψ, and $(u\sigma)_I = \beta(u)$, $(v\sigma)_I = \beta(v)$. With this, $D\sigma$ is in $\mathcal{K}_\Psi$. Therefore I satisfies $D\sigma$ so that $u\sigma_I = v\sigma_I$, and hence $\beta(u) = \beta(v)$. Now that F has been shown to weakly satisfy $\mathcal{K}$, according to the assumption there exists a total $\mathcal{K}$-algebra I' into which F can be weakly embedded. This algebra I' satisfies $s_i \approx t_i$, as F does, and since the embedding is injective, I' also satisfies $s \not\approx t$. Altogether, $I' \not\models C$, which contradicts the assumption that $\mathcal{K} \models C$. □

Hence we see that the weaker form of Evans' criterion with satisfaction replaced by weak satisfaction implies locality. For a large subclass of presentations, the distinction between the two forms of satisfaction is inessential. Lets us call a presentation $\mathcal{K}$ *superficial*, if every term that occurs positively (in the head) of a clause in $\mathcal{K}$ also occurs as a subterm negatively (in the body) the same clause.

THEOREM 6.3 Let $\mathcal{K}$ be a set of flat, linear, and superficial Horn clauses. Then $\mathcal{K}$ is local, whenever every finite partial $\mathcal{K}$-algebra weakly embeds into $\mathcal{K}$.

Proof. The definedness requirements for partial $\mathcal{K}$-algebras are void, if every positive functional term also appears negatively in the same clause. In that case, any partial algebra which weakly satisfies $\mathcal{K}$ is a partial $\mathcal{K}$-algebra, and the theorem follows from Theorem 6.2 □

For arbitrary presentations, the existence of weak embeddings for finite partial $\mathcal{K}$-algebras implies stable locality.

THEOREM 6.4 Let $\mathcal{K}$ be a set of Horn clauses. Suppose that every finite partial $\mathcal{K}$-algebra weakly embeds into $\mathcal{K}$. Then $\mathcal{K}$ is stably local.

Proof. Let C be a ground clause. We have to show that, under the given assumptions, if $\mathcal{K} \models C$, then $\mathcal{K}_{[C]} \models C$. Let $C = s_1 \approx t_1,\ldots,s_k \approx t_k \rightarrow s \approx t$, and let us assume, for the purpose of deriving a contradiction, that C is not entailed by $\mathcal{K}_{[C]}$. Then there exists an algebra I satisfying $\mathcal{K}_{[C]}$ and the equations $s_i \approx t_i$, but s and t are different in I, that is, I satisfies $s \not\approx t$. From this we will now construct a finite, partial $\mathcal{K}$-algebra F satisfying $s_i \approx t_i$ and $s \not\approx t$. The main difference to the proof of Theorem 6.2 will be that more terms are going to be defined in F.

Let $F = \{t_I \mid t \text{ a term in } \mathsf{st}[C]\}$, and let the functions f in Σ be defined such that I is an expansion of F. In other words, a function application $f_F(a_1,\ldots,a_n)$ in F is defined and yields a as result, iff $f_I(a_1,\ldots,a_n) = a$ with a in F. By construction, F satisfies the equations $s_i \approx t_i$ as well as $s \not\approx t$. Let $D = u_1 \approx v_1,\ldots,u_m \approx v_m \rightarrow u \approx v$ be a clause in $\mathcal{K}$ and let β be an assignment of elements in F to the variables. Then the pair D,β corresponds to at least one instance of D in $\mathcal{K}_{[C]}$. And, since function application $f_F(a_1,\ldots,a_n)$ in F is defined whenever the evaluation $f_I(a_1,\ldots,a_n)$ of the application in I yields a value in F, F satisfies $\mathcal{K}$ such that both conditions (i) and (ii) in the definition of satisfaction are met. In other words, F is a partial $\mathcal{K}$-algebra. Hence, there exists a total $\mathcal{K}$-algebra I' into which F can be embedded. This algebra I' satisfies the equations $s_i \approx t_i$, as F does, and since the embedding is injective, I' also satisfies $s \not\approx t$. Altogether, $I' \not\models C$, which contradicts the assumption that $\mathcal{K} \models C$. □

The preceding theorem, in connection with Theorem 5.1, strengthens Theorem 2.1 by also providing us with a concrete complexity bound. It also shows that Evans' approach is subsumed by stable locality. This subsumption is proper. Consider $\mathsf{Int}^{(4)}$ given as

$$\begin{aligned} p(x)\approx y, s(y)\approx z &\rightarrow z\approx x \\ s(x)\approx y, p(y)\approx z &\rightarrow z\approx x \end{aligned}$$

Like Int', the presentation $\mathsf{Int}^{(4)}$ is stably local. However, the two-element algebra $A = \{a,b\}$ with $s_A(a) = s_A(b) = a$ and $p_A = \emptyset$, the totally undefined function, trivially (strongly) satisfies $\mathsf{Int}^{(4)}$ but cannot be weakly embedded into $\mathsf{Int}^{(4)}$ where s has to be injective.

7 Locality and Axiomatizable Classes of Relational Substructures

Burris' results are based on the view of partial algebras as relational structures. Remember that for a signature Σ without predicate symbols, by Σ^* we denote the corresponding relational signature where each n-ary function symbol f in Σ is replaced by a $n+1$-ary relation symbol r^f. Σ^*-clauses are formed from the predicate symbols in Σ^*, the equality symbol, and variables. Similarly, if A is a Σ-algebra, by A^* we denote its relational variant, the Σ^*-structure for which $r^f_{A^*}(a_1,\ldots,a_n,a)$ if, and only if, $f_A(a_1,\ldots,a_n) = a$. If C is an equational Horn clause with all equations of the form $f(x_1,\ldots,x_k)\approx x$ or $x\approx y$, with variables x_i, x, y, by C^* we denote its relational variant where all equations $f(x_1,\ldots,x_k)\approx x$ are replaced by atoms $r^f(x_1,\ldots,x_k,x)$. If K is a class of total Σ-algebras, by $S(K^*)$ we denote the class of *full substructures* of members of K^*, that is the class of Σ^*-structures A^* for which there exists an algebra B in K such that B^* is an expansion of A^*. On the other hand, by $\overline{S}(K^*)$ we denote the class of *weak substructures* of members of K^*. This class coincides with the class of Σ^*-structures that weakly embed into K, that is, with $\{P^* \mid P \text{ weakly embeds into an algebra } A \in K\}$. By construction we have $S(\mathcal{K}^*) \subseteq \overline{S}(\mathcal{K}^*)$. (A full substructure is obtained by intersecting the graphs of the functions in a total algebra with the chosen subset of its carrier. Weak substructures are obtained from full substructures by making the functions even less defined. Hence there are more weak substructures than full substructures.)

THEOREM 7.1 (BURRIS 1995) Let $\mathcal{K}$ be a quasi-variety over Σ such that there is a finite set of Horn clauses H over Σ^* with $S(\mathcal{K}^*) \subseteq H \subseteq \overline{S}(\mathcal{K}^*)$. Then the uniform word problem for $\mathcal{K}$ is decidable in polynomial time.

The criterion says that if some subclass of relational weak substructures of $\mathcal{K}$ which includes all full substructures is finitely axiomatizable, the uniform word problem is decidable in polynomial time. We will show constructively that this criterion implies the existence of a stably local presentation, and that, conversely, from a local presentation a suitable H can be effectively constructed.

It is not surprising that in comparing Burris' criterion with locality we encounter the same technical problem with constants as we did in Section 6. Hence from now on in this section we restrict the classes $S(\mathcal{K}^*)$, H (the models of H), and $\overline{S}(\mathcal{K}^*)$ to structures in which the relations r^a are nonempty, for every constant a appearing in $\mathcal{K}$. In other words, if A is a partial algebra for which A^* is in any of these classes, we again require that the constants in $\mathcal{K}$ be defined in A.

Given a set H of Σ^*-clauses, by H_* we denote the set of equational Σ-clauses obtained from H by replacing atoms $r^f(x_1,\ldots,x_n,x)$ by equations $f(x_1,\ldots,x_k)\approx x$. Clearly, H_* is a flat set of clauses. Note that if A is a partial Σ-algebra, then A^* satisfies H if, and only if, A (strongly) satisfies H_*. A has to satisfy the definedness requirements implied by H_* in order for A^* to satisfy H. For example, if $p^a(x), p^b(y) \rightarrow r^f(x,y)$ is a clause in H, the corresponding clause in H_* will be $a\approx x, b\approx y \rightarrow f(x)\approx y$. In order for A^* to satisfy $p^a(x), p^b(y) \rightarrow r^f(x,y)$, f_A has to be defined on a_A with $f_A(a_A) = b_A$.

THEOREM 7.2 Let H be a set of Σ^*-clauses with $S(\mathcal{K}^*) \subseteq H \subseteq \overline{S}(\mathcal{K}^*)$. Then H_* is a stably local presentation of the quasi-variety $\mathcal{K}$.

Proof. First we show that the class of algebras satisfying H_* coincides with $\mathcal{K}$. If $A \models H_*$ then $A^* \models H$, and therefore, A^* is in $\overline{S}(\mathcal{K}^*)$. Hence A can be weakly embedded into an $\mathcal{K}$-algebra B. In other words, A is isomorphic to a $\mathcal{K}$-subalgebra, hence is a $\mathcal{K}$-algebra itself.

Conversely, suppose that A is in $\mathcal{K}$. Then A^* is in $S(\mathcal{K}^*)$, so that $A^* \models H$, hence, $A \models H_*$.

Now we show that H_* is stably local. According to Theorem 6.4 we have to show that every finite partial H_*-algebra A weakly embeds into H_*. Let such an algebra A be given. As $A^* \models H$ and as $H \subseteq \overline{S}(\mathcal{K}^*)$, the embedding property for A follows. □

Conversely, from a local theory $\mathcal{K}$ we can obtain a finite axiomatization $H_{\mathcal{K}}$ satisfying $S(\mathcal{K}^*) \subseteq H_{\mathcal{K}} \subseteq \overline{S}(\mathcal{K}^*)$. We may assume that $\mathcal{K}$ is flat and linear. (Otherwise, applying Proposition 4.1, we may replace $\mathcal{K}$ by $\mathcal{K}_{\mathsf{flin}}$, with $\mathcal{K}_{\mathsf{flin}}$ the set of flat and linear instances of $\mathcal{K}$.) Now define $H_{\mathcal{K}}$ to be the union of $\mathcal{K}^*$ and the set of uniqueness clauses $r^f(x_1,\ldots,x_n,y), r^f(x_1,\ldots,x_n,z) \rightarrow y\approx z$ for the relations.

THEOREM 7.3 With $H_{\mathcal{K}}$ as defined above, if $\mathcal{K}$ is a local theory, then $S(\mathcal{K}^*) \subseteq H_{\mathcal{K}} \subseteq \overline{S}(\mathcal{K}^*)$.

Proof. Clearly, all full substructures of $\mathcal{K}^*$ satisfy $H_{\mathcal{K}}$. Moreover, if $A^* \models H_{\mathcal{K}}$, then A is in particular a partial $\mathcal{K}$-

algebra. By Theorem 6.1, any such A weakly embeds into $\mathcal{K}$, hence A is in $\overline{S}(\mathcal{K}^*)$. □

8 Conclusion and Further Remarks

In this paper we have established close relationships between the approaches by Evans, Burris and McAllester to capture polynomial time computation in the context of uniform word problems. The criteria by Evans and Burris are essentially semantic, relating functional and relational models of given presentations. Local inference (McAllester's approach) and stable locality (our variant of this concept) are notions which are more proof-theoretic in nature. It was interesting to see how closely related these approaches are. We have shown that both Evans' and Burris' criteria lie in between the two variants of locality. The inclusions are proper (at least for Evans' approach, for Burris' we do not know yet). In particular the concept of stably local theories subsumes Burris' method (which in turn subsumes Evans' method), and the subsumption is strict for the Evans case.

8.1 Explicit Definedness Predicates

The reason why [stable] locality and the other two approaches are not quite equivalent is intimately related to the definedness requirements for partial functions that partial $\mathcal{K}$-algebras or full substructures of $\mathcal{K}^*$ have to satisfy. For the subclass of presentations for which the definedness requirements are void, we were able to establish equivalence of locality and Evans' criterion. Definedness, however, is a semantic concept that is not so easily captured syntactically. Only those clauses for which the antecedent is satisfiable contribute to definedness properties.

The following approach should work to simulate some of these effects in the framework of stable locality. Transform any given $\mathcal{K}$ by replacing clauses such as $s \approx t \rightarrow f(u) \approx g(v)$ by (read "$D(x)$" as "x is defined")

$$\begin{aligned} D(s), D(t), D(u), D(g(v)), s \approx t &\rightarrow D(f(u)) \\ D(s), D(t), D(u), D(f(u)), s \approx t &\rightarrow D(g(v)) \\ D(s), D(t), D(f(u)), D(g(v)), s \approx t &\rightarrow f(u) \approx g(v) \end{aligned}$$

internalizing the notion of satisfaction for partial algebras. (The general case of the transformation should be obvious.) Let $\mathcal{K}^D$ denote the result of that transformation. Call $\mathcal{K}$ "Evans"-local if for every ground clause $C = \Gamma \rightarrow e$ we have $\mathcal{K}^D[C] \cup \Gamma \cup D[C] \models e$ whenever $\mathcal{K} \models C$, with $D[C]$ the set of facts $D(t)$, for each subterm t in C and constant t in $\mathcal{K}$. Like in stable locality, we may use substitution instances of theory clauses where variables are sent to subterms of C or constants in $\mathcal{K}$. However, because of using the transformed clauses, we may additionally only compute with those terms in $\mathcal{K}_{[C]}$ that are *semantically* equal to a subterm of the query or to a constant in $\mathcal{K}$. Int′ is an example of a presentation that is Evans-local.

Evans-locality should be equivalent to Evans' criterion, but since its definition is somewhat awkward and since it is not clear how to design good recursively enumerable approximations of the concept, we have not yet investigated this claim in more detail.

8.2 Applications

It might be possible to exploit the relation between the semantic and proof-theoretic concepts for mutually transferring further techniques. In particular, certain (yet to be established) amalgamation properties for algebras would induce combination results for local theories over disjoint vocabularies.

Let P be any partial $\mathcal{K}$-algebra. Let $T(P) = T_\Sigma(P)/E_P$, that is, the free (total) Σ-term algebra generated over P, modulo the congruence E_P generated by the identities in P, that is, the equations $a_0 \approx f(a_1, \ldots, a_n)$ such that a_i is in P and $a_0 = f_P(a_1, \ldots, a_n)$ holds in P. We can turn $T(P)$ into a $\mathcal{K}$-algebra by dividing by the intersection K of all kernels of homomorphisms $h : T(P) \rightarrow A \in \mathcal{K}$. Let $F(P,\mathcal{K}) = T(P)/K$. A specific case of a much more general result proved by Burmeister (1986) is this universal property of $F(P,\mathcal{K})$:

THEOREM 8.1 (BURMEISTER 1986) $F(P,\mathcal{K})$ is a $\mathcal{K}$-algebra, and for any weak homomorphism h from P into a $\mathcal{K}$-algebra B there is a unique homomorphism $\overline{h} : F(P,\mathcal{K}) \rightarrow B$ such that $h = \overline{h} \circ \pi$, with $\pi : P \rightarrow F(P,\mathcal{K})$ the canonical weak homomorphism sending any element of P to its congruence class under K.

The theorem asserts that $F(P,\mathcal{K})$ is the free (total) $\mathcal{K}$-algebra generated by the partial Σ-algebra P.

COROLLARY 8.2 If P weakly embeds into $\mathcal{K}$ then the canonical weak homomorphism $\pi : P \rightarrow F(P,\mathcal{K})$ is injective.

Hence, whenever a partial algebra P weakly embeds into $\mathcal{K}$ it specifically also weakly embeds into its free extension $F(P,\mathcal{K})$, a fact already observed by Evans (1951).

Suppose now that we have two flat, linear, and superficial local theories $\mathcal{K}_1$ and $\mathcal{K}_2$ over disjoint signatures Σ_1 and Σ_2 respectively. We want to show that the union $\mathcal{K}_1 \cup \mathcal{K}_2$ is also local. One way to do this is to utilize the methods in (Nelson & Oppen 1979) for combining decision procedures. An algebraic proof might be obtained via an amalgamation construction similar to the one given by Baader & Schulz (1998) for proving decidability of the combination of certain unification problems.

We need to show that every finite partial $\mathcal{K}_1 \cup \mathcal{K}_2$-algebra P weakly embeds into $\mathcal{K}_1 \cup \mathcal{K}_2$, and then apply the theorem 6.3, Given P, forget the operations in Σ_2 and Σ_1, respectively, yielding a partial $\mathcal{K}_1$-algebra P_1 and a partial $\mathcal{K}_2$-algebra P_2. As the given theories are local, these algebras weakly embed into the free constructions $F(P_1,\mathcal{K}_1)$

and $F(P_2, \mathcal{K}_2)$, respectively. We believe that one can now amalgamate $F(P_1, \mathcal{K}_1)$ and $F(P_2, \mathcal{K}_2)$ into a single $\mathcal{K}_1 \cup \mathcal{K}_2$-algebra into which P weakly embeds to, but the details have not been worked out yet.

Acknowledgments I am are grateful to Sergei Vorobyov for directing my attention to Burris' paper and to Viorica Sofronie-Stokkermans for fruitful discussions.

References

Baader, F. & Schulz, K. (1998), 'Combination of constraint solvers for free and quasi-free structures', *Theoretical Computer Science* **192**, 107–161.

Bachmair, L. & Tiwari, A. (2000), Abstract congruence closure and specializations, *in* D. McAllester, ed., 'Automated Deduction – CADE-17, 17th International Conference on Automated Deduction', LNAI 1831, Springer-Verlag, Pittsburgh, PA, USA, pp. 64–78.

Basin, D. & Ganzinger, H. (2001), 'Automated complexity analysis based on ordered resolution', *J. Association for Computing Machinery* **48**(1), 70–109.

Burmeister, P. (1986), *A Model Theoretic Approach to Partial Algebras. Introduction to Theory and Application of Partial Algebras, Part. 1*, Akademie-Verlag, Berlin.

Burris, S. (1995), 'Polynomial time uniform word problems', *Mathematical Logic Quarterly* **41**, 173–182.

Cosmadakis, S. (1988), 'The word and generator problem for lattices', *Information and Computation* **77**, 192–217.

Dowling, W. F. & Gallier, J. H. (1984), 'Linear-time algorithms for testing the satisfiability of propositional Horn formulae', *J. Logic Programming* **3**, 267–284.

Downey, P. J., Sethi, R. & Tarjan, R. E. (1980), 'Variations on the common subexpressions problem', *J. Association for Computing Machinery* **27**(4), 771–785.

Evans, T. (1951), 'The word problem for abstract algebras', *J. London Math. Soc.* **26**, 64–71.

Freese, R. (1989), 'Finitely presented lattices: canonical forms and the covering relation', *Trans. Amer. Math. Soc.* **312**, 841–860.

Ganzinger, H. & McAllester, D. (2001), A new meta-complexity theorem for bottom-up logic programs, *in* 'Proc. International Joint Conference on Automated Reasoning', Lecture Notes in Computer Science, Springer-Verlag. To appear.

Ganzinger, H., Nieuwenhuis, R. & Nivela, P. (1994), The Saturate system. System available at http://www.mpi-sb.mpg.de/SATURATE/.

Givan, R. & McAllester, D. (1992), New results on local inference relations, *in* 'Principles of Knowledge Representation and reasoning: Proceedings of the Third International Conference (KR'92)', Morgan Kaufmann Press, pp. 403–412.

Knuth, D. & Bendix, P. (1970), Simple word problems in universal algebras, *in* J. Leech, ed., 'Computational Problems in Abstract Algebra', Pergamon Press, Oxford, pp. 263–297.

Kozen, D. (1977), Complexity of finitely presented algebras, *in* 'Proc. 9th STOC', pp. 164–177.

McAllester, D. (1993), 'Automatic recognition of tractability in inference relations', *J. Association for Computing Machinery* **40**(2), 284–303.

McAllester, D. (1999), On the complexity analysis of static analyses, *in* A. Cortesi & R. Filé, eds, 'Static Analysis — 6th International Symposium, SAS'99', LNCS 1694, Springer-Verlag, Venice, Italy, pp. 312–329.

Nelson, G. & Oppen, D. C. (1979), 'Simplification by cooperating decision procedures', *ACM Transactions on Programming Languages and Systems* **2**(2), 245–257.

Scott, D. (1979), Identity and existence in intuitionistic logic, *in* M. Fourman, ed., 'Durham proceedings (1977), Applications of Sheaves', Vol. 753 of *Lecture Notes in Mathematics*, Springer Verlag, pp. 660–696.

Skolem, T. (1920), 'Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit und Beweisbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen', *Skrifter utgitt av Videnskapsselskapet i Kristiania* **I**(4), 1–36. Also in: Th. Skolem, Selected Works in Logic (Jens Erik Fendstad, ed.), Scand. Univ. Books, Universitetsforlaget, Oslo, 1970, pp. 103–136.

Session 3

Semantics of Name and Value Passing

Marcelo Fiore*
Computer Laboratory
University of Cambridge

Daniele Turi†
LFCS, Division of Informatics
University of Edinburgh

Abstract

We provide a semantic framework for (first order) message-passing process calculi by combining categorical theories of abstract syntax with binding and operational semantics. In particular, we obtain abstract rule formats for name and value passing with both late and early interpretations. These formats induce an initial-algebra/final-coalgebra semantics that is compositional, respects substitution, and is fully abstract for late and early congruence. We exemplify the theory with the π-calculus and value-passing CCS.

Introduction

A complete description of the semantics of a programming language requires both an operational semantics describing the behaviour of programs in terms of elementary steps and a more abstract denotational semantics describing the meaning of a program in terms of its components [32]. In the study of process calculi for concurrency (such as CCS [25], CSP [19], and ACP [4]) less emphasis is placed on denotational models and more on notions of behavioural equivalence, and on bisimulation equivalence [25] in particular. Still, for the operational semantics to be well-behaved, one requires that the chosen notion of behavioural equivalence be a congruence with respect to the constructs of the language.

To establish congruence results for behavioural equivalences it is convenient to define the operational semantics in terms of structural rules, i.e., Plotkin's SOS rules [29]. Correspondingly, much work has been done in order to identify SOS rule formats [10, 6, 17, 14] for which (strong) bisimulation is a congruence – the most well-known being GSOS [6]. However, such formats are hard to find and even harder to extend. Little or no success at all has been gained, e.g., in obtaining formats for more sophisticated process calculi than the above mentioned ones – process calculi with variable binding (like value-passing CCS [26] and the π-calculus [27]) in particular. The present paper addresses this very problem.

The solution we offer is based on understanding the mathematical structure underlying syntax and semantics of message passing processes. The formats we obtain are abstract and require a fair amount of category theory. However, concrete, syntactic formats can be distilled from them and this, indeed, will be the next step of our investigation.

The starting point for our work lies in [35], where a categorical rule format is defined in terms of functorial notions Σ and B of syntax and behaviour familiar from initial algebra [16] and final coalgebra [1, 36] semantics. This format is given by transformations

$$\Sigma(X \times BX) \longrightarrow BTX \tag{1}$$

natural in the parameter X (to be thought of as a generic set of meta-variables used in the rules), where T is the term monad associated to the signature Σ, i.e., $TX = \mu Y.\, X + \Sigma Y$.

The type in (1) arises from giving to each operator of arity n of the signature a natural transformation

$$(X \times BX)^n \longrightarrow BTX \tag{2}$$

describing the overall behaviour of the operator in terms of the behaviour of its arguments. This abstract format corresponds to GSOS when B is taken to be the functor on **Set** whose coalgebras are finitely branching labelled transition systems, i.e.,

$$BX = \wp_{\mathrm{f}}(L \times X) \tag{3}$$

where L is a finite set of labels and $\wp_{\mathrm{f}}$ is the finite powerset functor. In this case, the domain $(X \times \wp_{\mathrm{f}}(L \times X))^n$ and the codomain $\wp_{\mathrm{f}}(L \times TX)$ of the map in (2) correspond, respectively, to the premises and the conclusions of GSOS rules for the operator. Interestingly, naturality accounts exactly for the GSOS restrictions on the occurrences of variables in the rules.

Any natural transformation of type (1) has the property that the coalgebraic behavioural equivalence associated to B (which in the above case coincides with bisimulation [2])

*Research supported by an EPSRC Advanced Research Fellowship.
†Research supported by EPSRC grant R34723.

0-7695-1281-X/01 $10.00 © 2001 IEEE

is a congruence with respect to the operators of the syntax Σ. This is a corollary of the more general fact that rules in the format (1) induce a denotational semantics which is adequate in the sense that it is fully abstract with respect to behavioural equivalence.

The above result is independent from the choice of category and functors, provided they have enough structure and properties. Here we exploit this generality in order to find formats for process calculi with variable binding. To this end, we first had to give a functorial notion of syntax with binding. This was one of the main motivations for the work in [13], where we moved from sets to variable sets. There, variable sets are taken to be functors (called covariant presheaves) from a category of contexts to **Set**; the category of contexts used is the category $\mathbb{F}$ of finite cardinals (i.e., sets of variables) and all functions (i.e., renamings). Most importantly, there exist a distinguished presheaf V of variables and a differentiation functor $\delta = (_)^V$ on presheaves. The latter is used to model variable binding with arity V: for a presheaf X, the elements of δX in context n are simply the elements of X in the context $n+1$ containing an extra variable – the variable to be bound.

We have now to find the right notions of behaviour B for name and value passing. Let us start from name passing, where the two most natural notions of behavioural equivalence are late and early bisimulation [27]. These are not congruences for the π-calculus though; one then considers the late and early congruences instead [27], obtained by closing bisimulation under renamings (i.e., the maps of $\mathbb{F}$).

Previous (implicitly) coalgebraic work on name passing [12, 33] was based on a functor B whose associated behavioural equivalence turns out to be late bisimulation. This functor B lives in the category of presheaves over the category $\mathbb{I}$ of name contexts allowing only injective renamings. Surprisingly, the natural extension of such B to the category of presheaves over $\mathbb{F}$ yields a new behaviour $\widetilde{B}$ whose associated equivalence is exactly late congruence.

We are also able to solve the problem left open in [12, 33] of giving a denotational semantics fully abstract with respect to early bisimulation by introducing a new behaviour whose associated equivalence is early bisimulation[1]. The extension of such behaviour to the presheaves over $\mathbb{F}$ has early congruence as associated equivalence. Therefore, the desired formats for early and late congruences live in the category of presheaves over $\mathbb{F}$ and, for instance, rules for unary binding will be of type

$$(X \times \widetilde{B}X)^V \longrightarrow \widetilde{B}TX \qquad (4)$$

where $\widetilde{B}$ can be the extended behaviour for either late or early congruence.

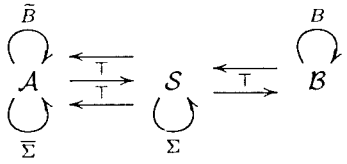
[1] See also [28] for a different coalgebraic approach to early (and late) bisimulation and [8] for a domain equation for early bisimulation in the framework of presheaf models.

For value passing, we also give late and early behaviours, which are variations (cf. [20]) of the behaviour in (3). However, in order to model input rules we have to take into account the substitution structure present in value-passing calculi, i.e., the homogeneous substitution of messages in messages and the heterogeneous substitution of messages in processes. (For name passing this is not needed because substitution is just renaming, hence it is already, though implicitly, part of the category of presheaves over $\mathbb{F}$.)

The categorical framework for homogeneous substitution was developed in [13]. One considers a monoidal structure on presheaves '$\bullet$' with unit V. A presheaf $X \bullet Y$ can be thought of has having elements given by pairs of an element of X together with a substitution consisting of a tuple of elements of Y. One then takes the notion of homogeneous substitution on a presheaf M to be a monoid structure $V \longrightarrow M \longleftarrow M \bullet M$.

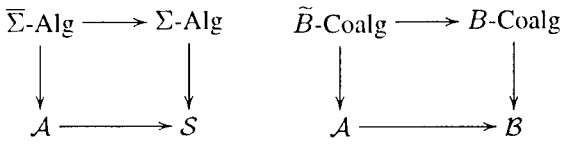
Here, in order to model the heterogeneous substitution of elements of a monoid M in elements of a presheaf X, we need to go one step further and consider monoid actions $X \bullet M \longrightarrow X$. Correspondingly, the modelling of rules takes place in the category of actions of the monoid of messages. Therefore, we need then to lift signatures with binding Σ and extend behaviours B to functors $\overline{\Sigma}$ and $\widetilde{B}$ on such category.

In general, we have primitive notions Σ and B living in different categories, of syntax $\mathcal{S}$ and behaviour $\mathcal{B}$ respectively, while the rules live in yet another category $\mathcal{A}$ of substitutions (e.g., monoid actions). These categories are related by adjunctions:



(5)

The lifting of the Σ on $\mathcal{S}$ to a $\overline{\Sigma}$ on $\mathcal{A}$ is done by means of a distributive law over the monad induced by the monadic adjunction $\mathcal{A} \rightleftarrows \mathcal{S}$, while the behaviour $\widetilde{B}$ on $\mathcal{A}$ is obtained by (right) extending B on $\mathcal{B}$ along the composite adjunction $\mathcal{A} \rightleftarrows \mathcal{B}$. These constructions yield liftings/extensions as follows:



The abstract rule format ensuring that behavioural equivalence is a congruence consists then of natural transformations of type

$$\overline{\Sigma}(X \times \widetilde{B}X) \longrightarrow \widetilde{B}\overline{T}X \qquad (6)$$

For name passing the actions of the monoid of variables are simply presheaves on $\mathbb{F}$, hence $\overline{\Sigma}$ is equal to Σ. For the original GSOS case of [35], with no variable binding, all three categories collapse to the category of sets, hence $\overline{\Sigma}$ and $\widetilde{B}$ are equal to Σ and B respectively and we recover (1).

The next obvious step for our work is to characterise the categorical rule formats for name and value passing proposed in this paper in elementary syntactic terms. The rule formats so obtained will certainly not be as in [5], where binding and substitution are defined within the rules rather than treated at the syntactic level. For value passing, our categorical rule format seems to be related to a syntactic format proposed in [30]. The relationship with the format of [15] for which a conservative extension property holds should also be investigated.

Another aspect we would like to consider is recursion. At present we would deal with guarded recursion following [34], but it would be interesting to deal with unguarded recursion along the lines of [31], hence working with variable cpos instead of variable sets.

Finally, there seems to be a tight correspondence between the coalgebras of our new behaviour for early bisimulation and the indexed labelled transition systems of [7]. We would like to investigate this for sheaves (in the Schanuel topos) rather than presheaves over $\mathbb{I}$.

1. Basic syntactic and semantic structures

1.1. Expressions

Syntax. Consider the following abstract grammar of expressions for integers

$$e ::= x \mid \underline{z} \mid e_1 \,\mathtt{plus}\, e_2 \mid e_1 \,\mathtt{minus}\, e_2 \tag{7}$$

where x ranges over a countable list of variables x_i ($i \in \mathbb{N}$) and $\underline{z}$ over the set of integers $\mathbb{Z}$.

Following [13], we consider terms in a context, so that we can stratify expressions into a family $\{ E_n \}_{n\in\mathbb{N}}$ of sets indexed by natural numbers (indicating the number of variables in the context). The set E_n consists of the expressions with at most n (*canonical*) free variables (typically denoted by $x_1, \ldots, x_n$). Thus, $\{ E_n \}_{n\in\mathbb{N}}$ is the least solution of the equations

$$\{ X_n = \{ x_1, \ldots, x_n \} + \mathbb{Z} + {X_n}^2 + {X_n}^2 \}_{n\in\mathbb{N}} \tag{8}$$

Semantics. We write $\mathcal{E}[\![e]\!]_n$ for the interpretation of an expression e in the context $x_1, \ldots, x_n$; that is, for the function $\mathbb{Z}^n \longrightarrow \mathbb{Z}$ defined compositionally as follows:

$$\begin{array}{l} 1.\ \mathcal{E}[\![x_i]\!]_n = \pi_i \quad (i^{\text{th}} \text{ projection}, 1 \le i \le n) \\ 2.\ \mathcal{E}[\![\underline{z}]\!]_n = \lambda\vec{x}.z \quad (\text{constant function } z) \\ 3.\ \mathcal{E}[\![e_1 \,\mathtt{plus}\, e_2]\!]_n = \lambda\vec{x}.(\mathcal{E}[\![e_1]\!]_n(\vec{x}) + \mathcal{E}[\![e_2]\!]_n(\vec{x})) \\ 4.\ \mathcal{E}[\![e_1 \,\mathtt{minus}\, e_2]\!]_n = \lambda\vec{x}.(\mathcal{E}[\![e_1]\!]_n(\vec{x}) - \mathcal{E}[\![e_2]\!]_n(\vec{x})) \end{array} \tag{9}$$

This interpretation is an initial algebra semantics. Indeed, the semantic domain given by

$$\{ \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \}_{n\in\mathbb{N}} \tag{10}$$

where $\mathbf{Set}(S, S')$ denotes the set of functions from a set S to a set S', has a (pointwise) algebra structure given by the evident maps

$$\begin{array}{l} \{ x_1, \ldots, x_n \} \longrightarrow \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \\ \mathbb{Z} \longrightarrow \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \\ \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z})^2 \cong \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}^2) \longrightarrow \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \\ \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z})^2 \cong \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}^2) \longrightarrow \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \end{array} \tag{11}$$

and

$$\mathcal{E} = \{ \mathcal{E}[\![_]\!]_n : E_n \longrightarrow \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \}_{n\in\mathbb{N}} \tag{12}$$

is the unique algebra homomorphism from $\{ E_n \}_{n\in\mathbb{N}}$ to $\{ \mathbf{Set}(\mathbb{Z}^n, \mathbb{Z}) \}_{n\in\mathbb{N}}$.

1.2. Presheaves

Categorically, families $\{ X_n \}_{n\in\mathbb{N}}$ of sets are functors

$$X : \mathbb{N} \longrightarrow \mathbf{Set}$$

where $\mathbb{N}$ is the discrete category of natural numbers or, equivalently, finite cardinals. Since we regard a finite cardinal n as a context of n variables, a function $\rho : n \longrightarrow m$ can be seen as a renaming of variables. In order to model weakening, contraction, and exchange rules for contexts we need to use, instead of the discrete category $\mathbb{N}$, the category $\mathbb{F}$ of finite cardinals and all functions (cf. [13]). Correspondingly, we consider functors

$$X : \mathbb{F} \longrightarrow \mathbf{Set}$$

i.e., (covariant) *presheaves* over $\mathbb{F}$. Thus, we will be working with families $\{ X_n \}_{n\in\mathbb{N}}$ of sets equipped with an *action* that associates every $x \in X_n$ (i.e., an element of X at stage n) and every renaming $\rho : n \longrightarrow m$ with

$$x[\rho] = X(\rho)(x) \in X_m$$

Presheaves over $\mathbb{F}$ form a category $\mathbf{Set}^{\mathbb{F}}$, with natural transformations as morphisms.

Syntax. The family $\{ E_n \}_{n\in\mathbb{N}}$ with action

$$e[\rho] = e[{}^{x_{\rho 1}}/_{x_1}, \ldots, {}^{x_{\rho n}}/_{x_n}] \qquad (\rho : n \longrightarrow m)$$

given by variable renaming defines a presheaf $E : \mathbb{F} \to \mathbf{Set}$. This presheaf is the least solution of the equation

$$X = V + \mathcal{K}_{\mathbb{Z}} + X^2 + X^2$$

in $\mathbf{Set}^{\mathbb{F}}$ (cf. (8)), where the presheaf of *variables*

$$V : \mathbb{F} \longrightarrow \mathbf{Set}\,, \qquad V_n = n \cong \{ x_1, \ldots, x_n \}$$

is the inclusion of $\mathbb{F}$ into $\mathbf{Set}$ and $\mathcal{K}_{\mathbb{Z}}$ is the constantly $\mathbb{Z}$ presheaf. Hence E is the free Σ-algebra $\mu Y.\, V + \Sigma Y$ over the presheaf of variables V, where

$$\Sigma : \mathbf{Set}^{\mathbb{F}} \longrightarrow \mathbf{Set}^{\mathbb{F}}\,, \quad \Sigma X = \mathbb{Z} + X^2 + X^2$$

is the endofunctor on presheaves associated to the operators on expressions.

Semantics. Also the semantic domain for expressions (10) has a presheaf structure. Indeed, for any object C of a cartesian category $\mathcal{C}$, we have a functor

$$\langle C, _ \rangle : \mathcal{C} \longrightarrow \mathbf{Set}^{\mathbb{F}}\,, \qquad \langle C, D \rangle_n = \mathcal{C}(C^n, D) \qquad (13)$$

The presheaf $\langle C, D \rangle$ can be thought of as the presheaf of mappings from environments of type C to results of type D. Formally, at stage n, it consists of the set of morphisms in $\mathcal{C}$ from C^n to D with action

$$f[\rho] = f \circ \langle \pi_{\rho 1}, \ldots, \pi_{\rho n} \rangle \qquad (\rho : n \longrightarrow m)$$

In particular, taking $\mathcal{C} = \mathbf{Set}$ and $C = D = \mathbb{Z}$ we obtain the presheaf $\langle \mathbb{Z}, \mathbb{Z} \rangle$ with underlying family of sets as in (10).

The copairing of the maps in (11) gives a Σ-algebra structure

$$\Sigma\langle \mathbb{Z}, \mathbb{Z} \rangle = \mathcal{K}_{\mathbb{Z}} + \langle \mathbb{Z}, \mathbb{Z} \rangle^2 + \langle \mathbb{Z}, \mathbb{Z} \rangle^2 \longrightarrow \langle \mathbb{Z}, \mathbb{Z} \rangle \qquad (14)$$

on $\langle \mathbb{Z}, \mathbb{Z} \rangle$ that induces the initial algebra semantics

$$\mathcal{E} : E \longrightarrow \langle \mathbb{Z}, \mathbb{Z} \rangle$$

of (12). Note that the naturality of $\mathcal{E}$ amounts to the identity

$$\begin{aligned} &\mathcal{E}[\![\, e[{}^{x_{\rho 1}}/_{x_1}, \ldots, {}^{x_{\rho n}}/_{x_n}] \,]\!]_m(z_1, \ldots, z_m) \\ &\quad = \mathcal{E}[\![e]\!]_n(z_{\rho 1}, \ldots, z_{\rho n}) \end{aligned} \qquad (15)$$

for all $\rho : n \longrightarrow m$.

Syntax with binding. In the algebraic treatment of binding of [13], binding operators are modelled using the *differentiation* operator

$$\delta : \mathbf{Set}^{\mathbb{F}} \longrightarrow \mathbf{Set}^{\mathbb{F}}\,, \qquad (\delta X)_n = X_{n+1}$$

(For details, including initial algebra semantics, consult [13].)

Pi-calculus. The following grammar for (a fragment of) the π-calculus

$$t ::= \mathbf{0} \mid t_1 | t_2 \mid x(y).t \mid \bar{x}y.t \mid (x)t \mid [x = y]\, t$$

corresponds to the signature endofunctor

$$\begin{aligned} \Sigma X \;=\; & 1 + X \times X + V \times \delta X + V \times V \times X \\ & + \delta X + V \times V \times X \end{aligned} \qquad (16)$$

on $\mathbf{Set}^{\mathbb{F}}$. Indeed, its initial algebra

$$\begin{aligned} T0 \;\cong\; & 1 + T0 \times T0 + V \times \delta T0 + V \times V \times T0 \\ & + \delta T0 + V \times V \times T0 \end{aligned}$$

is the presheaf of π-calculus terms: at stage n it is the set of (α-equivalence classes of) terms with at most n (canonical) free variables, with action given by variable renaming.

Value-passing CCS. We will consider the following fragment of CCS passing expressions e as in (7) along a finite set of channels $c \in C$:

$$t ::= \mathbf{0} \mid t_1 | t_2 \mid c?(x).t \mid c!\langle e \rangle.t \mid [e_1 = e_2]\, t$$

This grammar has associated signature endofunctor

$$\begin{aligned} \Sigma_E X \;=\; & 1 + X \times X + \mathcal{K}_C \times \delta X \\ & + \mathcal{K}_C \times E \times X + E \times E \times X \end{aligned}$$

on $\mathbf{Set}^{\mathbb{F}}$, where $\mathcal{K}_C$ is the constantly C presheaf.

More generally, we have a signature bifunctor $\Sigma : \mathbf{Set}^{\mathbb{F}} \times \mathbf{Set}^{\mathbb{F}} \longrightarrow \mathbf{Set}^{\mathbb{F}}$

$$\begin{aligned} \Sigma(M, X) \;=\; & 1 + X \times X + \mathcal{K}_C \times \delta X \\ & + \mathcal{K}_C \times M \times X + M \times M \times X \end{aligned} \qquad (17)$$

parametric in the presheaf of messages being passed.

1.3. Substitution

Clones. We have seen that besides the operators, the semantics $\mathcal{E}$ also respects variable renaming (see (9) and (15)). However, $\mathcal{E}$ respects *substitution* in the stronger form of satisfying the *semantic substitution lemma*:

$$\begin{aligned} &\mathcal{E}[\![e\,[{}^{e_1}/_{x_1}, \ldots, {}^{e_n}/_{x_n}]]\!]_m \\ &\quad = \mathcal{E}[\![e]\!]_n \circ \langle \mathcal{E}[\![e_1]\!]_m, \ldots, \mathcal{E}[\![e_n]\!]_m \rangle \end{aligned} \qquad (18)$$

In other words, $\mathcal{E}$ is not only an algebra homomorphism but also, as we explain below, a clone homomorphism.

Recall that an *(abstract) clone* [9, page 132] X, consists of a family $\{ X_n \}_{n\in\mathbb{N}}$ of sets, a family

$$\{ \nu_i^{(n)} \in X_n \mid 1 \leq i \leq n \}_{n\in\mathbb{N}}$$

of distinguished elements, and a family

$$\{ \mu_m^{(n)} : X_n \times (X_m)^n \longrightarrow X_m \}_{n,m\in\mathbb{N}}$$

of operations such that, for every element t of X_n, every n-tuple $\vec{u} = (u_1, \ldots, u_n)$ of elements of X_m, and every m-tuple $\vec{v}$ of elements of X_ℓ, the following three axioms hold:

$$\begin{aligned} &\mu_m(\nu_i; \vec{u}) = u_i \qquad \mu_n(t; \nu_1, \ldots, \nu_n) = t \\ &\mu_\ell(\mu_m(t; \vec{u}); \vec{v}) = \mu_\ell(t; \mu_\ell(u_1; \vec{v}), \ldots, \mu_\ell(u_n; \vec{v})) \end{aligned} \tag{19}$$

An homomorphism $h : X \longrightarrow X'$ between clones is a family $\{ h_n : X_n \longrightarrow X'_n \}_{n\in\mathbb{N}}$ of functions that respects the clone structure.

The clone structure on the family $\{ E_n \}_{n\in\mathbb{N}}$ of expressions is given by the variables x_i $(1 \leq i \leq n)$ in E_n and by the simultaneous substitution of expressions for expressions

$$\begin{array}{ccc} E_n \times (E_m)^n & \longrightarrow & E_m \\ (e; e_1, \ldots, e_n) & \longmapsto & e[^{e_1}/_{x_1}, \ldots, ^{e_n}/_{x_n}] \end{array}$$

(The three axioms in (19) amount to the familiar properties of substitution.) For the semantic domain $\langle \mathbb{Z}, \mathbb{Z} \rangle$, the clone structure is given by projections and function composition (together with pairing). In fact, for every object C of a cartesian category $\mathcal{C}$, one can form the *clone of operations* $\langle C, C \rangle$ on C, with $\nu_i^{(n)}$ given by the i^{th} projection $\pi_i : C^n \longrightarrow C$ and $\mu_m^{(n)}$ by the map

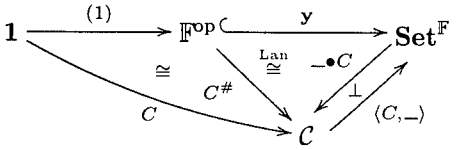
$$\begin{array}{ccc} \mathcal{C}(C^n, C) \times \mathcal{C}(C^m, C)^n & \longrightarrow & \mathcal{C}(C^m, C) \\ (f; f_1, \ldots, f_n) & \longmapsto & f \circ \langle f_1, \ldots, f_n \rangle \end{array}$$

Thus, with respect to the above clone structures, the requirement that the semantics $\mathcal{E}$ be a clone homomorphisms amounts to the identity (9.1) and the semantic substitution lemma (18).

Monoids. The clone structure has equivalent representations as either of the following: finitary monads on **Set**, Lawvere theories, substitution algebras [13, Theorem 3.3], or, most importantly for this work, monoids in the monoidal closed category $(\mathbf{Set}^{\mathbb{F}}, \bullet, V)$ [13, Proposition 3.4], where the monoidal product is defined by the following coend:

$$(X \bullet Y)_m = \int^{n\in\mathbb{F}} X_n \times (Y_m)^n \qquad (m \in \mathbb{F}) \tag{20}$$

This tensor product and variations thereof play a crucial role in this paper; they arise from the following general situation (see, e.g., [23, I.5]):



(21)

where $\mathcal{C}$ is cartesian and cocomplete and where $C^{\#}$ denotes the cartesian extension of C.

Proposition 1.1 1. For $\mathcal{C}$ and $\mathcal{D}$ cartesian and cocomplete categories, and $F : \mathcal{C} \longrightarrow \mathcal{D}$ a cartesian functor with a right adjoint, we have a canonical natural isomorphism

$$_ \bullet FC \cong F(_ \bullet C)$$

for all $C \in \mathcal{C}$.

2. For a cartesian and cocomplete category $\mathcal{C}$ such that, for all $C \in \mathcal{C}$, the functor $_ \times C$ is cocontinuous, we have the following equivalence of categories

$$\begin{array}{ccc} \mathcal{C} & \simeq & \mathsf{CarCoc}(\mathbf{Set}^{\mathbb{F}}, \mathcal{C}) \\ C & \longmapsto & _ \bullet C \\ FV & \longleftarrow & F \end{array}$$

where CarCoc is the category of cartesian and cocontinuous functors, and natural transformations. □

Corollary 1.2 For every $X \in \mathbf{Set}^{\mathbb{F}}$ and $C \in \mathbf{Set}^{\mathbb{C}}$, there are canonical natural isomorphisms as follows

$$\begin{array}{rcl} (_ \bullet X) \bullet C & \cong & _ \bullet (X \bullet C) \\ \langle X, \langle C, _ \rangle \rangle & \cong & \langle X \bullet C, _ \rangle \end{array}$$ □

In this paper we will exclusively consider the above tensor construction when $\mathcal{C} = \mathbf{Set}^{\mathbb{C}}$, for some small category $\mathbb{C}$ (see [23, VII.2 and VIII.4] for a general discussion in the context of topos theory). In this case, the tensor $X \bullet C$ (for $X \in \mathbf{Set}^{\mathbb{F}}$ and $C \in \mathbf{Set}^{\mathbb{C}}$) has the following elementary description

$$\begin{aligned} (X \bullet C)_m &= \textstyle\int^{n\in\mathbb{F}} X_n \times (C_m)^n \\ &\cong (\coprod_{n\in\mathbb{N}} X_n \times (C_m)^n)_{/\approx} \end{aligned} \qquad (m \in \mathbb{C})$$

where $\approx$ is the equivalence relation generated by

$$(x; c_{\rho 1}, \ldots, c_{\rho n}) \sim (x[\rho]; c_1, \ldots, c_{n'}) \quad (\rho : n \twoheadrightarrow n')$$

Note that in particular taking $\mathbb{C} = \mathbb{F}$ and $C = Y \in \mathbf{Set}^{\mathbb{F}}$ we obtain the tensor (20) on $\mathbf{Set}^{\mathbb{F}}$. We will also use the case where $\mathbb{C} = \mathbf{1}$ (the terminal category), hence $\mathcal{C} \cong \mathbf{Set}$ and C is a set S:

$$X \bullet S = \int^{n\in\mathbb{F}} X_n \times S^n$$

As mentioned above, the categories of clones and monoids in $(\mathbf{Set}^{\mathbb{F}}, \bullet, V)$ are equivalent, hence the semantics $\mathcal{E} : E \longrightarrow \langle \mathbb{Z}, \mathbb{Z} \rangle$ is both a Σ-algebra homomorphism and a monoid homomorphism. In fact, by Theorem 4.1 of [13], the presheaf of expressions E is the initial object in the category of Σ-*monoids* (consisting of compatible Σ-algebra and monoid structures with corresponding homomorphisms). And, as the Σ-algebra structure in (14) for the clone of operations $\langle \mathbb{Z}, \mathbb{Z} \rangle$ is compatible with the clone/monoid structure of $\langle \mathbb{Z}, \mathbb{Z} \rangle$, the semantics $\mathcal{E}$ is the unique Σ-monoid homomorphism from E to $\langle \mathbb{Z}, \mathbb{Z} \rangle$.

1.4. Categorical operational semantics

It is shown in [35] that operational rules of the form (1) for signature and behaviour endofunctors Σ and B on a bicartesian category $\mathcal{C}$ induce a compositional semantics having the (full abstraction) property that two terms have the same meaning if and only if they are bisimilar, provided that (*i*) the forgetful functor B-Coalg $\longrightarrow \mathcal{C}$ has a right adjoint (hence a final coalgebra exists), and (*ii*) the behaviour B preserves weak pullbacks. The main tool we use to establish (*i*) for the behaviours in the present paper is the following.

Proposition 1.3 (See [24, 3]) For a finitary (resp. accessible) endofunctor B on a locally finitely presentable (resp. accessible) category $\mathcal{B}$, the forgetful functor B-Coalg $\longrightarrow \mathcal{B}$ has a right adjoint. □

The above mentioned (coalgebraic) notion of *bisimulation* is due to [2]. In this paper, we will consider it in the following form: a *B-bisimulation* between two coalgebras $h : X \longrightarrow BX$ and $k : Y \longrightarrow BY$ is a relation (i.e., equivalence class of monos) $R \hookrightarrow X \times Y$ between the carriers X and Y which lifts to the coalgebras in the sense that the diagram

$$\begin{array}{ccccc} X & \longleftarrow & R & \longrightarrow & Y \\ \downarrow & & \downarrow & & \downarrow \\ BX & \longleftarrow & BR & \longrightarrow & BY \end{array}$$

commutes for some coalgebra structure on R. For the behaviour in (3) B-bisimulation is (strong) bisimulation.

2. Message passing bisimulations

2.1. Value passing

Late bisimulation. To model value-passing CCS, with respect to a set of values $\mathbb{V}$ and a *finite* set of channels C, we consider the behaviour endofunctor

$$BS = \wp_{\mathrm{f}}(C \times S^{\mathbb{V}} + C \times \mathbb{V} \times S + S) \qquad (22)$$

on $\mathbf{Set}$, where the components of the sum respectively model input, output, and silent actions. (Cf. [20].)

With respect to this behaviour functor, coalgebraic bisimulation corresponds to late bisimilarity. Indeed, a coalgebra $h : S \longrightarrow BS$ induces the late transition relation

$$s \xrightarrow{c?()} f \text{ iff } (c, f) \in h(s) \quad (c \in C, s \in S, f \in S^{\mathbb{V}})$$

$$s \xrightarrow{c!\langle v \rangle} s' \text{ iff } (c, v, s') \in h(s) \quad (c \in C, v \in \mathbb{V}, s, s' \in S)$$

$$s \xrightarrow{\tau} s' \text{ iff } s' \in h(s) \quad (s, s' \in A)$$

that provides a characterisation of coalgebraic bisimulation in familiar terms (see [21]) as follows.

Proposition 2.1 The following data are equivalent.

1. A coalgebraic bisimulation for a coalgebra on S.
2. A symmetric relation $R \subseteq S \times S$ such that $s_0 \; R \; s_0'$ implies
   - if $s_0 \xrightarrow{c?()} f$ then there exists f' such that $s_0' \xrightarrow{c?()} f'$ and $f(v) \; R \; f'(v)$ for all $v \in \mathbb{V}$;
   - if $s_0 \xrightarrow{c!\langle v \rangle} s$ then there exists s' such that $s_0' \xrightarrow{c!\langle v \rangle} s'$ and $s' \; R \; s_0'$;
   - if $s_0 \xrightarrow{\tau} s$ then there exists s' such that $s_0' \xrightarrow{\tau} s'$ and $s \; R \; s'$. □

To appreciate the way in which (22) models the late interpretation of input, it is instructive to use the isomorphism $\wp_{\mathrm{f}}(S + S') \cong \wp_{\mathrm{f}}(S) \times \wp_{\mathrm{f}}(S')$ and consider the behaviour in the following form

$$BS \cong \wp_{\mathrm{f}}(S^{\mathbb{V}})^C \times \wp_{\mathrm{f}}(\mathbb{V} \times S)^C \times \wp_{\mathrm{f}}(S)$$

from which, as observed by Gordon Plotkin, one can read the late interpretation off the first component of the product corresponding to "first choosing a derivative and then receiving a value". To model the early interpretation of input, corresponding to "first receiving a value and then choosing a derivative", one thus needs to reverse the role of the type constructors for non-determinism and inaction, and input.

Early bisimulation. Noticing the following decomposition of the finite powerset functor

$$\wp_{\mathrm{f}} \cong 1 + \wp_{\mathrm{f}}^{+}$$

where $\wp_{\mathrm{f}}^{+}$ is the *non-empty* finite powerset functor, a natural behaviour for the early interpretation is then the endofunctor

$$BS = (1 + \wp_{\mathrm{f}}^{+}(S)^{\mathbb{V}})^C \times \wp_{\mathrm{f}}(\mathbb{V} \times S)^C \times \wp_{\mathrm{f}}(S)$$

which we will consider below in the following uniform form

$$\begin{aligned} BS \;\cong\; & (C \rightharpoonup \wp_{\mathrm{f}}^{+}(S)^{\mathbb{V}}) \\ & \times (C \rightharpoonup \wp_{\mathrm{f}}^{+}(\mathbb{V} \times S)) \\ & \times (1 \rightharpoonup \wp_{\mathrm{f}}^{+}(S)) \end{aligned} \tag{23}$$

where $_ \rightharpoonup _ : p\mathbf{Set}^{\mathrm{op}} \times p\mathbf{Set} \longrightarrow \mathbf{Set}$ is the *partial-exponential* functor (see e.g. [11]).

In this setting, a coalgebra $h : S \longrightarrow BS$ induces the early transition relation

$s \xrightarrow{c?\langle v\rangle} s'$ iff $s' \in \pi_1(hx)(c)(v)$ $(c \in C, v \in \mathbb{V}, s, s' \in S)$

$s \xrightarrow{c!\langle v\rangle} s'$ iff $(v, s') \in \pi_2(hx)(c)$ $(c \in C, v \in \mathbb{V}, s, s' \in S)$

$s \xrightarrow{\tau} s'$ iff $s' \in \pi_3(hx)()$ $(s, s' \in S)$

that provides a characterisation of coalgebraic bisimulation in familiar terms as follows.

Proposition 2.2 The following data are equivalent.

1. A coalgebraic bisimulation for a coalgebra on S.
2. A symmetric relation $R \subseteq S \times S$ such that $s_0 \; R \; s_0'$ implies
   - if $s_0 \xrightarrow{c?\langle v\rangle} s$ then there exists s' such that $s_0' \xrightarrow{c?\langle v\rangle} s'$ and $s \; R \; s'$;
   - if $s_0 \xrightarrow{c!\langle v\rangle} s$ then there exists s' such that $s_0' \xrightarrow{c!\langle v\rangle} s'$ and $s \; R \; s'$;
   - if $s_0 \xrightarrow{\tau} s$ then there exists s' such that $s_0' \xrightarrow{\tau} s'$ and $s \; R \; s'$. □

2.2. Name passing

Following [12], we will consider notions of behaviour for the π-calculus in the category of (variable sets) $\mathbf{Set}^{\mathbb{I}}$, where $\mathbb{I}$ is the category of finite cardinals and injections. However, all the constructions involved are also meaningful for pullback-preserving presheaves in $\mathbf{Set}^{\mathbb{I}}$ and so, following [33], we also obtain notions of behaviour in the Schanuel topos (see e.g. [23, pages 155 and 158]).

Late bisimulation. The constructions needed to model late bisimulation [27] as in [12] are:

- The type of *names* $N \in \mathbf{Set}^{\mathbb{I}}$ with identity action $N_n = n$.
- The *power* type $\wp_{\mathrm{f}} : \mathbf{Set}^{\mathbb{I}} \longrightarrow \mathbf{Set}^{\mathbb{I}}$ with pointwise action $(\wp_{\mathrm{f}} P)_n = \wp_{\mathrm{f}}(P_n)$.
- *Products* $(\times)$ and *coproducts* $(+)$ given pointwise by $(P \times Q)_n = P_n \times Q_n$ and $(P + Q)_n = P_n + Q_n$.
- The *exponential* P^N with action given by $(P^N)_n = (P_n)^n \times P_{n+1}$ and $P(\iota)(f, p) = (f', p')$ where

$$f'(x) = \begin{cases} (fa)[\iota] & \text{if } x = \iota a \\ p[\iota, x] & \text{otherwise} \end{cases} \quad \text{and} \quad p' = p[\iota + 1]$$

- The *dynamic allocation* type $\delta : \mathbf{Set}^{\mathbb{I}} \longrightarrow \mathbf{Set}^{\mathbb{I}}$ with action given by $(\delta P)_n = P_{n+1}$ and $(\delta P)(\iota) = P(\iota + 1)$.

The behaviour functor for late bisimulation of [12, 33] is

$$BP = \wp_{\mathrm{f}}(N \times P^N + N \times N \times P + N \times \delta P + P) \tag{24}$$

on $\mathbf{Set}^{\mathbb{I}}$. Hence we have that

$$\begin{aligned} BP_n = \; \wp_{\mathrm{f}}(\; & n \times (P_n)^n \times P_{n+1} \\ & + n \times n \times P_n \; + \; n \times P_{n+1} \\ & + P_n \qquad\qquad\qquad\qquad) \end{aligned}$$

in $\mathbf{Set}$.

A coalgebra $h : P \longrightarrow BP$ induces the late transition relation

$p \xrightarrow{a?()} f, p'$ iff $(a, f, p') \in h_n(p)$
$(a \in n, p \in P_n, f \in (P_n)^n, p' \in P_{n+1})$

$p \xrightarrow{a!\langle b\rangle} p'$ iff $(a, b, p') \in h_n(p)$
$(a, b \in n, p, p' \in P_n)$

$p \xrightarrow{a!()} p'$ iff $(a, p') \in h_n(p)$
$(a \in n, p \in P_n, p' \in P_{n+1})$

$p \xrightarrow{\tau} p'$ iff $p' \in h_n(p)$
$(p, p' \in P_n)$

that provides a characterisation of coalgebraic bisimulation in familiar terms (see [27]) as follows.

Proposition 2.3 The following data are equivalent.

1. A coalgebraic bisimulation for a coalgebra on P.
2. A family of symmetric relations
$$\{ R_n \subseteq P_n \times P_n \}_{n \in \mathbb{N}}$$
such that, for every $n \in \mathbb{N}$,
   (a) $p \; R_n \; q$ implies $p[\iota] \; R_m \; q[\iota]$, for all $\iota : n \rightarrowtail m$ in $\mathbb{I}$;
   (b) $p \; R_n \; q$ implies
   - if $p \xrightarrow{a?()} f, p'$ then there exist g, q' such that $q \xrightarrow{a?()} g, q'$, and $f(a) \; R_n \; g(a)$ (for all $a \in n$) and $p' \; R_{n+1} \; q'$;
   - if $p \xrightarrow{a!\langle b\rangle} p'$ then there exists q' such that $q \xrightarrow{a!\langle b\rangle} q'$ and $p' \; R_{n+1} \; q'$;

- if $p \xrightarrow{a!()} p'$ then there exists q' such that $q \xrightarrow{a!()} q'$ and $p' \; R_{n+1} \; q'$;
- if $p \xrightarrow{\tau} p'$ then there exists q' such that $p' \xrightarrow{\tau} q'$ and $p' \; R_n \; q'$. □

Early bisimulation. The definition of a behaviour functor for early bisimulation (left open in [12, 33]) requires the introduction of a new type constructor.

- For a *mono-preserving* presheaf $P : \mathbb{I} \longrightarrow \mathbf{Set}$ we define $P \Rightarrow _ : \mathbf{Set}^{\mathbb{I}} \longrightarrow \mathbf{Set}^{\mathbb{I}}$ as the functor mapping a presheaf Q to the presheaf $P \Rightarrow Q$ with action given by $(P \Rightarrow Q)_n = P_n \Rightarrow Q_n$ and

$$(P \Rightarrow Q)(\iota) = P(\iota) \Rightarrow Q(\iota) : u \longmapsto Q(\iota) \circ u \circ P(\iota)^R$$

where $P(\iota)^R(q) = p$ iff $P(\iota)(p) = q$ (see [11]).

This construction extends that of products in that we have an injection $P \times Q \rightarrowtail P \Rightarrow Q$ given by:

$$\begin{array}{ccc} P_n \times Q_n & \rightarrowtail & P_n \Rightarrow Q_n \\ p, q & \longmapsto & p^R \Rightarrow q \end{array} \tag{25}$$

where $(p^R \Rightarrow q)(x) = (\text{if } x = p \text{ then } q)$.

In the vein of the treatment of early bisimulation for value-passing CCS given in (23), we consider the following behaviour functor

$$\begin{aligned} BP = {} & (N \Rightarrow \wp_{\mathrm{f}}^{+}(P)^N) \\ & \times (N \Rightarrow \wp_{\mathrm{f}}^{+}(N \times P)) \times (N \Rightarrow \wp_{\mathrm{f}}^{+}(\delta P)) \\ & \times (1 \Rightarrow \wp_{\mathrm{f}}^{+}(P)) \end{aligned} \tag{26}$$

in $\mathbf{Set}^{\mathbb{I}}$, where the components of the product respectively model input, free and bound output, and silent actions. (The role of the constructor $N \Rightarrow _$ in this behaviour functor is analogous to the one of the topped tensor product $N \otimes^{\top} _$ in the model of [18].)

Note that because of the following isomorphisms

$$\begin{aligned} \wp_{\mathrm{f}}(P + Q) &\cong \wp_{\mathrm{f}}(P) \times \wp_{\mathrm{f}}(Q) \\ \wp_{\mathrm{f}}(N \times P) &\cong N \Rightarrow \wp_{\mathrm{f}}^{+}(P) \\ \wp_{\mathrm{f}}(P) &\cong 1 \Rightarrow \wp_{\mathrm{f}}^{+}(P) \end{aligned}$$

the *late* behaviour functor (24) can be written in the following form

$$\begin{aligned} & (N \Rightarrow \wp_{\mathrm{f}}^{+}(P^N)) \\ & \times (N \Rightarrow \wp_{\mathrm{f}}^{+}(N \times P)) \times (N \Rightarrow \wp_{\mathrm{f}}^{+}(\delta P)) \\ & \times (1 \Rightarrow \wp_{\mathrm{f}}^{+}(P)) \end{aligned}$$

which makes clear that the late and early interpretations of free and bound output, and of silent actions are the same.

Considering the pointwise early behaviour

$$\begin{aligned} BP_n = {} & (n \Rightarrow (\wp_{\mathrm{f}}^{+} P_n)^n \times \wp_{\mathrm{f}}^{+} P_{n+1}) \\ & \times (n \Rightarrow \wp_{\mathrm{f}}^{+}(n \times P_n)) \\ & \times (n \Rightarrow \wp_{\mathrm{f}}^{+} P_{n+1}) \\ & \times (1 \Rightarrow \wp_{\mathrm{f}}^{+} P_n) \end{aligned}$$

a coalgebra $h : P \longrightarrow BP$ induces the early transition relation

$$p \xrightarrow{a?\langle b \rangle} p' \text{ iff } p' \in \pi_1(\pi_1(h_n p)(a))(b) \qquad (a, b \in n, p, p' \in P_n)$$

$$p \xrightarrow{a?()} p' \text{ iff } p' \in \pi_2(\pi_1(h_n p)(a)) \qquad (a \in n, p \in P_n, p' \in P_{n+1})$$

$$p \xrightarrow{a!\langle b \rangle} p' \text{ iff } (b, p') \in \pi_2(h_n p)(a) \qquad (a, b \in n, p, p' \in P_n)$$

$$p \xrightarrow{a!()} p' \text{ iff } p' \in \pi_3(h_n p)(a) \qquad (a \in n, p \in P_n, p' \in P_{n+1})$$

$$p \xrightarrow{\tau} p' \text{ iff } p' \in \pi_4(h_n p)() \qquad (p, p' \in P_n)$$

that provides a characterisation of coalgebraic bisimulation in familiar terms (see [27]) as follows.

Proposition 2.4 The following data are equivalent.

1. A coalgebraic bisimulation for a coalgebra on P.
2. A family of symmetric relations

$$\{ R_n \subseteq P_n \times P_n \}_{n \in \mathbb{N}}$$

such that, for every $n \in \mathbb{N}$,

(a) $p \; R_n \; q$ implies $p[\iota] \; R_m \; q[\iota]$, for all $\iota : n \rightarrowtail m$ in $\mathbb{I}$;

(b) $p \; R_n \; q$ implies

- if $p \xrightarrow{a?\langle b \rangle} p'$ then there exists q' such that $q \xrightarrow{a?\langle b \rangle} q'$ and $p' \; R_n \; q'$;
- if $p \xrightarrow{a?()} p'$ then there exists q' such that $q \xrightarrow{a?()} q'$ and $p' \; R_{n+1} \; q'$;
- if $p \xrightarrow{a!\langle b \rangle} p'$ then there exists q' such that $q \xrightarrow{a!\langle b \rangle} q'$ and $p' \; R_n \; q'$;
- if $p \xrightarrow{a!()} p'$ then there exists q' such that $q \xrightarrow{a!()} q'$ and $p' \; R_{n+1} \; q'$;
- if $p \xrightarrow{\tau} p'$ then there exists q' such that $q \xrightarrow{\tau} q'$ and $p' \; R_n \; q'$. □

3 Semantics of name passing

To model the structural operational rules for the π-calculus using natural transformations of type (1), we are faced with the fact that the signature Σ is an endofunctor on $\mathbf{Set}^{\mathbb{F}}$ (see (16)) while the behaviour B (for both the late (24) and the and the early (26) interpretations) is an endofunctor on $\mathbf{Set}^{\mathbb{I}}$. Far from being a problem, this disparity allows for the desired compositionality result to hold. Indeed, both late and early bisimulations are *not* congruences. What we need are thus behaviour functors for late and early *congruences* instead. These behaviours can be obtained by (right) extending the B's on $\mathbf{Set}^{\mathbb{I}}$ along an adjunction $\mathbf{Set}^{\mathbb{F}} \underset{T}{\overleftarrow{\longrightarrow}} \mathbf{Set}^{\mathbb{I}}$ obtaining new endofunctors $\widetilde{B}$'s on $\mathbf{Set}^{\mathbb{F}}$. Moreover, a natural transformation of type

$$\Sigma(X \times \widetilde{B}X) \longrightarrow \widetilde{B}TX \tag{27}$$

in $\mathbf{Set}^{\mathbb{F}}$ will be suitable to model the desired structural operational rules for the π-calculus.

Late and early congruences. The adjunction we need between $\mathbf{Set}^{\mathbb{F}}$ and $\mathbf{Set}^{\mathbb{I}}$ is an instance of the adjunction in (21) taking $\mathcal{C} = \mathbf{Set}^{\mathbb{I}}$ and $C = N$:

$$\mathbf{Set}^{\mathbb{F}} \underset{-\bullet N}{\overset{\langle N,_\rangle}{\underset{T}{\leftarrow}}} \mathbf{Set}^{\mathbb{I}} \tag{28}$$

Alternatively, one can describe this adjunction as the essential geometric morphism (see, e.g., [23, page 360]) associated to the inclusion $\mathbb{I} \longrightarrow \mathbb{F}$. Thus, we have a canonical natural isomorphism

$$X \bullet N \cong |X| \tag{29}$$

(essentially given by the action $X_n \times m^n \longrightarrow X_m$ of X) where $|_| : \mathbf{Set}^{\mathbb{F}} \longrightarrow \mathbf{Set}^{\mathbb{I}}$ is the *forgetful* functor given by precomposing with the inclusion $\mathbb{I} \longrightarrow \mathbb{F}$.

We can now define, for every endofunctor B on $\mathbf{Set}^{\mathbb{I}}$, an endofunctor

$$\widetilde{B}X = \langle N, B|X|\rangle$$

i.e., the *right Kan extension* of $\langle N, B_\rangle$ along $\langle N, _\rangle$. Using the isomorphism (29) and the adjunction (28), the $\widetilde{B}$-coalgebras are in bijective correspondence with B-coalgebras $|X| \longrightarrow B|X|$. In other words, $\widetilde{B}$-coalgebras are B-coalgebras on presheaves with an action along *all* renamings (rather than only on injective ones). This makes a crucial difference in terms of coalgebraic bisimulation.

Proposition 3.1 For B as in (24) [resp. (26)], the following data are equivalent.

1. A coalgebraic $\widetilde{B}$-bisimulation for a coalgebra $X \longrightarrow \widetilde{B}X$.
2. A family of symmetric relations $\{R_n \subseteq X_n \times X_n\}_{n\in\mathbb{N}}$ as in Proposition 2.3 (2) [resp. Proposition 2.4 (2)] (with respect to the transposed B-coalgebra $|X| \longrightarrow B|X|$) where the closure condition (a) is generalised to

   $p\ R_n\ q$ implies $p[\rho]\ R_m\ q[\rho]$, for all $\rho : n \longrightarrow m$ in $\mathbb{F}$. □

Proposition 3.2 1. The functors $(_)^N : \mathbf{Set}^{\mathbb{I}} \longrightarrow \mathbf{Set}^{\mathbb{I}}$ and $N^n \pitchfork _ : \mathbf{Set}^{\mathbb{I}} \longrightarrow \mathbf{Set}^{\mathbb{I}}$ $(n \in \mathbb{N})$ are finitary.

2. For B as in (24) and (26), the lifted functors $\widetilde{B}$ are finitary (hence the forgetful functor $\widetilde{B}$-Coalg $\longrightarrow \mathbf{Set}^{\mathbb{F}}$ has a right adjoint) and preserve weak pullbacks. □

Therefore, every natural transformation of type (27), with B the late (early) behaviour functor, induces a compositional semantics fully abstract with respect to late (early) congruence.

Categorical rules. We sketch how the π-calculus operational rules [27] are modelled by a natural transformation of type (27). For brevity, we only consider the operational rules of the binding operators (input and restriction); the operational rules for the other operators are modelled along the lines of [34] using the isomorphisms

$$\langle C, D_1\rangle \times \langle C, D_2\rangle \cong \langle C, D_1 \times D_2\rangle$$
$$\delta\langle C, D\rangle \cong \langle C, D^C\rangle$$

satisfied by the functors in (13) with $\mathcal{C}$ cartesian closed, and the map

$$V \times X \longrightarrow \langle N, N \pitchfork |X|\rangle \tag{30}$$

obtained by transposing $|V \times X| \cong N \times |X| \rightarrowtail N \pitchfork |X|$, where the injection is given by (25).

Input. For input, the rule is modelled by a map of type

$$V \times \delta(X \times \widetilde{B}X) \longrightarrow \langle N, B|TX|\rangle$$

Using (30) and projecting out the components that do not contribute to the rule we can focus on defining a map of type

$$\delta X \longrightarrow \langle N, |X|^N\rangle \cong \delta\langle N, |X|\rangle$$

The required map is δ applied to the unit $X \rightarrowtail \langle N, |X|\rangle$ of the adjunction (28); that is,

$$\begin{array}{rcl} X_{n+1} & \longrightarrow & \mathbf{Set}^{\mathbb{I}}(N^{n+1}, |X|) \\ x & \longmapsto & \{\, \lambda\rho \in m^{n+1}.\, x[\rho]\,\}_{m\in\mathbb{I}} \end{array}$$

Note that this map can be used both for the late and early cases by precomposing it with suitable maps respectively arising from the injections $|X|^N \rightarrowtail \wp^+_{\mathrm{f}}(|X|^N)$ and $|X|^N \rightarrowtail (\wp^+_{\mathrm{f}}|X|)^N$.

Restriction. For restriction, the rule is modelled by a map of type $\delta(X \times \widetilde{B}X) \longrightarrow \langle N, B|TX|\rangle$ in $\mathbf{Set}^{\mathbb{F}}$ which, in fact, comes from a map of type

$$\delta B|X| \longrightarrow B|TX| \quad \text{in } \mathbf{Set}^{\mathbb{I}}$$

For instance, the core of this latter map corresponding to the following two rules

$$\text{(RES)}\ \frac{\mathrm{P} \xrightarrow{\bar{a}b} \mathrm{Q}}{(x)\mathrm{P} \xrightarrow{\bar{a}b} (x)\mathrm{Q}}\ x \not\equiv a,b \qquad \text{(OPEN)}\ \frac{\mathrm{P} \xrightarrow{\bar{a}x} \mathrm{Q}}{(x)\mathrm{P} \xrightarrow{\bar{a}(x)} \mathrm{Q}}\ x \not\equiv a$$

is the map

$$\delta N \times \delta N \times \delta|X| \xrightarrow{\mathrm{RO}} \wp_{\mathrm{f}}(N \times N \times |TX| + N \times \delta|TX|)$$

defined, using the internal language (see [12]), as follows:

$$\begin{array}{l}
\mathrm{RO}(a,b,q) \\
\quad = \ \mathsf{case}\ a\ \mathsf{of} \\
\qquad \mathsf{old}(a') \ \Rightarrow\ \mathsf{let}\ q' = \delta\,\boldsymbol{\eta}\,q \\
\qquad\qquad\qquad \mathsf{in}\ \mathsf{case}\ b\ \mathsf{of} \\
\qquad\qquad\qquad\qquad \mathsf{old}(b') \ \Rightarrow\ \{\,(a', b', \boldsymbol{\nu}\,q')\,\} \\
\qquad\qquad\qquad\qquad\quad \mathsf{new} \ \Rightarrow\ \{\,(a', q')\,\} \\
\qquad\quad \mathsf{new} \ \Rightarrow\ \emptyset
\end{array}$$

where $\boldsymbol{\eta} : |X| \longrightarrow |TX|$ and $\boldsymbol{\nu} : \delta|TX| \longrightarrow |TX|$ (in $\mathbf{Set}^{\mathbb{I}}$) are respectively the (underlying maps of the) unit and the restriction operator (in $\mathbf{Set}^{\mathbb{F}}$) of the free Σ-algebra TX on X.

4. Semantics of value passing

Actions. We have seen in §1.1 that the homogeneous substitution of expressions for variables in expressions can be modelled as monoids. For the heterogeneous substitution of expressions for variables in terms we can use *monoid actions* as follows. Every monoid $M = (M, \mu, \nu)$ in $\mathbf{Set}^{\mathbb{F}}$ defines a monad $_ \bullet M$ on $\mathbf{Set}^{\mathbb{F}}$. The category of algebras of this monad M-**Act**, consists of (*right*) *actions* $A \bullet M \longrightarrow A$ [22, VII.4]. In elementary terms, this amounts to a family $\{\, \alpha_m^{(n)} : A_n \times (M_m)^n \longrightarrow A_m \,\}_{n,m \in \mathbb{N}}$ of operations such that

$$\alpha_m(a; \nu_1, \ldots, \nu_n) = a$$

$$\alpha_\ell(\alpha_m(a; \vec{u}); \vec{v}) = \alpha_\ell(a; \mu_\ell(u_1; \vec{v}), \ldots, \mu_\ell(u_n; \vec{v}))$$

for all a in A_n, $\vec{u}$ in $(M_m)^n$, and $\vec{v}$ in $(M_\ell)^m$. (Note the occurrence of μ in the second law.)

For examples of actions consider the following.

A V-action $A \bullet V \longrightarrow A$ is forced, by the unit law, to be the canonical isomorphism $A \bullet V \cong A$. Thus, the category V-**Act** is isomorphic to $\mathbf{Set}^{\mathbb{F}}$; which explains why, for name passing, we can do without extra substitution structure.

For objects C and D in a cartesian category $\mathcal{C}$, the monoid $\langle C, C\rangle$ has a canonical action on the presheaf $\langle C, D\rangle$ given by (pairing and) composition in $\mathcal{C}$.

As in any bicomplete monoidal closed category (cf. [23, VII.3]), a monoid homomorphism $M' \longrightarrow M$ induces a *reindexing* functor M-**Act** $\longrightarrow M'$-**Act** with both left and right adjoints. Thus, the semantics of expressions $E \longrightarrow \langle \mathbb{Z}, \mathbb{Z}\rangle$ and the unique homomorphism $V \longrightarrow M$ induce the following adjoint situations

$$\langle \mathbb{Z}, \mathbb{Z}\rangle\text{-}\mathbf{Act} \underset{\top}{\overset{\top}{\rightleftarrows}} E\text{-}\mathbf{Act}\,, \qquad M\text{-}\mathbf{Act} \underset{_\bullet M}{\overset{\langle M, _\rangle}{\underset{\top}{\overset{\top}{\rightleftarrows}}}} \mathbf{Set}^{\mathbb{F}}$$

where, on the right hand side, $X \bullet M$ has action given by multiplication and $\langle M, X\rangle$ has action given by multiplication and evaluation.

Syntax. The substitution of expressions in terms involves, in turn, a substitution of expressions in expressions. Thus, the signature bifunctor for value-passing CCS needs to be parametric in a *monoid* of messages. Accordingly, we let Σ be the bifunctor $\mathbf{Mon}(\mathbf{Set}^{\mathbb{F}}) \times \mathbf{Set}^{\mathbb{F}} \longrightarrow \mathbf{Set}^{\mathbb{F}}$ given by (17).

For a monoid M, we write Σ_M for the functor $\Sigma(M, _) : \mathbf{Set}^{\mathbb{F}} \longrightarrow \mathbf{Set}^{\mathbb{F}}$. One can lift Σ_M to the category M-**Act** of M-actions by means of a distributive law

$$\lambda : \Sigma_M(_) \bullet M \Longrightarrow \Sigma_M(_ \bullet M)$$

of the endofunctor Σ_M over the monad induced by the monadic adjunction M-**Act** $\underset{\top}{\overset{\top}{\rightleftarrows}}$ $\mathbf{Set}^{\mathbb{F}}$. This distributive law is essentially the strength described in [13, page 200], with the extra use of the multiplication of the monoid M in the fourth and fifth summand of Σ_M. The resulting endofunctor

$$\begin{array}{l}
\overline{\Sigma}_M(A \bullet M \xrightarrow{\alpha} A) \\
\qquad = \left(\Sigma_M(A) \bullet M \xrightarrow{\lambda_A} \Sigma_M(A \bullet M) \xrightarrow{\Sigma\alpha} \Sigma A \right)
\end{array}$$

on M-**Act** has as algebras presheaves A with both a Σ-algebra structure and an M-action compatible with each other in the sense that the evident diagram

$$\begin{array}{ccccc}
\Sigma_M(A) \bullet M & \longrightarrow & \Sigma_M(A \bullet M) & \longrightarrow & \Sigma_M(A) \\
\downarrow & & & & \downarrow \\
A \bullet M & & \longrightarrow & & A
\end{array}$$

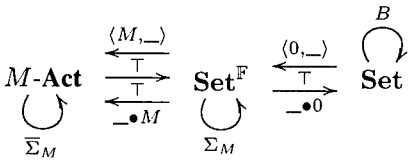
commutes. We denote the corresponding category of $\overline{\Sigma}_M$-algebras by $\overline{\Sigma}_M$-Alg. The associated forgetful functor $\overline{\Sigma}_M$-Alg $\longrightarrow M$-**Act** has a left adjoint; and the induced

monad is denoted by $\overline{T}_M$, as it is a lifting of the monad T_M induced by Σ_M.

Moreover, every monoid homomorphism $M' \longrightarrow M$ induces a reindexing functor $\overline{\Sigma}_M$-Alg $\longrightarrow \overline{\Sigma}_{M'}$-Alg, which is a lifting of the reindexing functor M-**Act** $\longrightarrow M'$-**Act**. In particular, the reindexing functor $\overline{\Sigma}_{\langle\mathbb{Z},\mathbb{Z}\rangle}$-Alg $\longrightarrow \overline{\Sigma}_E$-Alg induced by the semantics of expressions $E \longrightarrow \langle\mathbb{Z},\mathbb{Z}\rangle$ allows us to turn every interpretation for $\overline{T}_{\langle\mathbb{Z},\mathbb{Z}\rangle}(0)$ into one for $\overline{T}_E(0)$.

Semantics. Let M be a monoid of messages in $\mathbf{Set}^{\mathbb{F}}$; a typical example being the clone of operations $\langle\mathbb{V},\mathbb{V}\rangle$ on a set of values $\mathbb{V}$.

We have the following situation (cf. (5))



where the adjunction on the right can be alternatively described as the essential geometric morphism associated to the functor $(0) : \mathbf{1} \longrightarrow \mathbb{F}$; hence

$$X \bullet 0 \cong X_0$$

for all $X \in \mathbf{Set}^{\mathbb{F}}$.

To have both syntax and behaviour on the same category, we will proceed as in the previous section and (right) extend behaviour functors B on **Set** along the composite adjunction M-**Act** $\underset{\longrightarrow}{\overset{\longleftarrow}{\top}}$ **Set** to $\widetilde{B}$ on M-**Act**. To do this easily, we need a lemma.

Lemma 4.1 For $\mathcal{C}$ cartesian and cocomplete, the composite adjunction M-**Act** $\underset{|_|}{\overset{\langle M,_\rangle}{\top}}$ $\mathbf{Set}^{\mathbb{F}}$ $\underset{_\bullet C}{\overset{\langle C,_\rangle}{\top}}$ $\mathcal{C}$ is given by

$|_| \bullet C : M\text{-}\mathbf{Act} \underset{\longrightarrow}{\overset{\longleftarrow}{\top}} \mathcal{C} : \langle M \bullet C, _\rangle$. □

It follows that the extension of a behaviour functor B on **Set** is along the adjunction

$$|_|_0 : M\text{-}\mathbf{Act} \underset{\longrightarrow}{\overset{\longleftarrow}{\top}} \mathbf{Set} : \langle M_0, _\rangle \qquad (31)$$

where M_0 is the set of ground messages, yielding $\widetilde{B}$ on M-**Act** to be given by

$$\widetilde{B}A = \langle M_0, B(A_0)\rangle$$

Late and early congruences. As operational models for value passing we take $\widetilde{B}$-coalgebras

$$A \longrightarrow \langle M_0, B(A_0)\rangle$$

in M-**Act** where B is either of the two endofunctors on **Set** of (22) and (23). The adjunction (31) allows us to express these operational models in terms of coalgebras on **Set**. Indeed, they are in bijective correspondence with functions

$$A_0 \longrightarrow B(A_0)$$

where A carries an M-action. Moreover, $\widetilde{B}$-coalgebra homomorphisms are action homomorphisms which at stage 0 are also B-coalgebra homomorphisms:

$$\begin{array}{ccc} A \bullet M & \longrightarrow & A \\ \downarrow & & \downarrow \\ A' \bullet M & \longrightarrow & A' \end{array} \qquad \begin{array}{ccc} A_0 & \longrightarrow & B(A_0) \\ \downarrow & & \downarrow \\ A'_0 & \longrightarrow & B(A'_0) \end{array}$$

Proposition 4.2 For B as in (22) [resp. (23)], the following data are equivalent.

1. A coalgebraic $\widetilde{B}$-bisimulation for a coalgebra $A \longrightarrow \widetilde{B}A$.
2. A family of symmetric relations $\{R_n \subseteq A_n \times A_n\}_{n\in\mathbb{N}}$ such that
   (a) R_0 is as in Proposition 2.1 (2) [resp. Proposition 2.2 (2)] (with respect to the transposed B-coalgebra $A_0 \longrightarrow B(A_0)$).
   (b) For every $n \in \mathbb{N}$, $s\ R_n\ s'$ implies
   $$\alpha_m(s;\vec{v})\ R_m\ \alpha_m(s';\vec{v}), \text{ for all } \vec{v} \text{ in } (M_m)^n. \quad \square$$

Proposition 4.3 1. The category of actions M-**Act** is locally finitely presentable.

2. For B as in (22) and (23), the extended functors $\widetilde{B}$ are accessible (hence the forgetful functor $\widetilde{B}$-Coalg $\longrightarrow M$-**Act** has a right adjoint) and preserve weak pullbacks. □

Categorical rules. Natural transformations in M-**Act** of type

$$\overline{\Sigma}(A \times \widetilde{B}A) \longrightarrow \widetilde{B}\overline{T}A \qquad (32)$$

with B the late (early) behaviour functor with set of values $\mathbb{V} = M_0$, are suitable to model structural operational rules for languages with value passing and give a categorical format inducing fully-abstract compositional semantics with respect to late (early) congruence.

Input. The most interesting rule to model is the axiom for input. As for the π-calculus, the core of this rule (both for the late and early behaviour) lies in the map

$$\overline{\delta}A \longrightarrow \langle \mathbb{V}, A_0{}^{\mathbb{V}}\rangle \cong \overline{\delta}\langle \mathbb{V}, A_0\rangle$$

obtained by applying $\overline{\delta}$ to the unit of the adjunction (31), namely:

$$\begin{array}{ccc} A_{n+1} & \longrightarrow & \mathbf{Set}(\mathbb{V}^{n+1}, A_0) \\ a & \longmapsto & \lambda\vec{v} \in \mathbb{V}^{n+1}.\, \alpha_0(a;\vec{v}) \end{array}$$

Acknowledgements. We gratefully acknowledge discussions with Gian-Luca Cattani, Gordon Plotkin and Davide Sangiorgi.

References

[1] P. Aczel. *Non-well-founded sets*. Number 14 in Lecture Notes. CSLI, 1988.

[2] P. Aczel and P. F. Mendler. A final coalgebra theorem. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Proc. Category Theory and Computer Science*, volume 389 of *LNCS*, pages 357–365. Springer-Verlag, 1989.

[3] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*, volume 189 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1994.

[4] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[5] K. Bernstein. A congruence theorem for structured operational semantics of higher-order languages. In *Proc. 13th LICS Conf.*, pages 153–164. IEEE, Computer Society Press, 1998.

[6] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.

[7] G.-L. Cattani and P. Sewell. Models for name-passing processes: Interleaving and causal. In *Proc. 15th LICS Conf.*, pages 322–333. IEEE, Computer Society Press, 2000.

[8] G.-L. Cattani, I. Stark, and G. Winskel. Presheaf models for the π-calculus. In E. Moggi and G. Rosolini, editors. *Proc. Category Theory and Computer Science*, volume 1290 of *LNCS*, pages 106–126. Springer-Verlag, 1997.

[9] P. Cohn. *Universal Algebra*. Harper & Row, 1965.

[10] R. de Simone. Higher level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[11] M. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations Series. Cambridge University Press, 1996.

[12] M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus. In *Proc. 11th LICS Conf.*, pages 43–54. IEEE, Computer Society Press, 1996. (Full version to appear in *Information and Computation*).

[13] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. 14th LICS Conf.*, pages 193–202. IEEE, Computer Society Press, 1999.

[14] W. Fokkink and R. van Glabbeek. Ntyft/ntyxt rules reduce to ntree rules. *Information and Computation*, 126(1):1–10, 1996.

[15] W. Fokkink and C. Verhoef. A conservative look at operational semantics with variable binding. *Information and Computation*, 146(1):24–54, 1998.

[16] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume IV, pages 80–149. Prentice Hall, 1978.

[17] J. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.

[18] M. Hennessy. A fully abstract denotational semantics for the π-calculus. Technical Report 4, COGS, University of Sussex, 1996. To appear in *Theoretical Computer Science*.

[19] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[20] A. Ingólfsdóttir. A semantic theory for value-passing processes, Late approach, Part I: A denotational model and its complete axiomatization. Report Series RS-95-3, BRICS, Department of Computer Science, University of Aarhus, 1995.

[21] A. Ingólfsdóttir. A semantic theory for value-passing processes, Late approach, Part II: A behavioural semantics and full abstractness. Report Series RS-95-22, BRICS, Department of Computer Science, University of Aarhus, 1995.

[22] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[23] S. Mac Lane and I. Moerdijk. *Sheaves in geometry and logic: A First Introduction to Topos Theory*. Springer-Verlag, 1992.

[24] M. Makkai and R. Paré. *Accessible Categories: The Foundations of Categorical Model Theory*, volume 104 of *Contemporary Math*. Amer. Math. Soc., 1989.

[25] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[26] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[27] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, Sept. 1992.

[28] U. Montanari and M. Pistore. π-calculus, structured coalgebras and minimal HD-automata. In *Mathematical Foundations of Computer Science 2000*, volume 1893 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[29] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, 1981.

[30] G. Plotkin. Binding algebras: A step from universal algebra to type theory. Slides of a lecture at RTA'98, 1998.

[31] G. Plotkin. Bialgebraic semantics and recursion. Invited talk at *4th Workshop on Coalgebraic Methods in Computer Science*, Genova, Italy, 2001.

[32] D. Scott. Outline of a mathematical theory of computation. In *Proc. 4th Annual Princeton Conference on Inf. Sciences and Systems*, pages 169–176, 1970.

[33] I. Stark. A fully abstract domain model for the π-calculus. In *Proc. 11th LICS Conf.*, pages 36–42. IEEE, Computer Society Press, 1996.

[34] D. Turi. Categorical modelling of structural operational rules: case studies. In E. Moggi and G. Rosolini, editors, *Proc. Category Theory and Computer Science*, volume 1290 of *LNCS*, pages 127–146. Springer-Verlag, 1997.

[35] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proc. 12th LICS Conf.*, pages 280–291. IEEE, Computer Society Press, 1997.

[36] D. Turi and J. Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8(5):481–540, 1998.

A Fully Abstract Game Semantics of Local Exceptions

J. Laird
COGS, University of Sussex
jiml@cogs.susx.ac.uk

Abstract

A fully abstract game semantics for an extension of Idealized Algol with locally declared exceptions is presented. It is based on "Hyland-Ong games", but as well as relaxing the constraints which impose functional behaviour (as in games models of other computational effects such as continuations and references), new structure is added to plays in the form of additional pointers which track the flow of control. The semantics is proved to be fully abstract by a factorization of strategies into a 'new-exception generator' and a strategy with local control flow. It is shown, using examples, that there is no model of exceptions which is a conservative extension of the semantics of Idealized Algol without the new pointers.

1 Introduction

All practical programming languages provide some means of manipulating the flow of control, primarily to recover from errors and deal with other exceptional eventualities. Dynamically bound, locally declared exceptions are a simple, elegant and effective way to do this, making them a key part of ML and Java, for example. Despite their ease of use for programmers, however, these exceptions are not 'easy' from a semantic point of view; no denotational model of a language containing them has hitherto been described. *Statically bound* exceptions can be implemented using call-with-current-continuation, but fail to account for one of the most important features of exceptions — that the same error may be handled in different ways if it occurs in different contexts On the other hand, dynamically bound *global* exceptions have been modelled abstractly via the exceptions monad [13], but this approach has not been applied to locally exceptions. It may be argued that local exceptions have proved resistant to the efforts of semanticists in part because they are a kind of *hybrid* effect. Their main purpose is to give access to the flow of control, but dynamic binding distinguishes them from statically bound control constructs such as **call/cc**, whilst locality gives rise to some of the identity-related issues which appear with reference variables. But since continuations and store have traditionally been modelled by (very different) constructions, simply piling them on top of a functional basis is likely to lead to a complicated semantics which is not fully abstract.

The basis for a possible solution to these problems can be found in the 'intensional hierarchy' [4] of games models of various effects such as state [1, 3], first-class continuations [11] and higher type references [5]. These all extend the basic model of PCF described by Hyland and Ong [9], and Nickau [14], by relaxing, one-by-one, the constraints on games and strategies which oblige them to behave in a purely functional way. This 'direct' approach to modelling side-effects means that they can often be combined simply (and fully abstractly) by relaxing the relevant combination of constraints.

However, even in the context of game semantics, the dynamic nature of exceptions has significant ramifications. Rather than simply weakening the appropriate constraints on the model of PCF, it proves necessary to to add significant new structure — in the form of additional 'contingency pointers' — to the traces which represent the states of a game, in order to describe the dynamic binding of exceptions. These pointers give an explicit representation of control flow in the model, allowing a move to be played as if it immediately follows an earlier move which is not actually its immediate predecessor.

The main contribution of this paper is therefore to define a new category of games by adding contingency pointers to HO-style games, to show that this category contains a model of locally declared, dynamically bound exceptions, and — by a full abstraction result — to show precisely how these combine locality with manipulation of control-flow. Using this analysis, it will show that the contingency structure really is necessary to interpret exceptions in HO games. Because the

0-7695-1281-X/01 $10.00 © 2001 IEEE

games models of references and continuations do not have this structure, this suggests that exceptions cannot be expressed using continuations and references.

Acknowledgements

The work reported here was supported by several UK EPSRC grants. I am grateful to Guy McCusker in particular for his comments.

2 Idealized Exceptions

The language which will be modelled — Idealized Algol [16] with (idealized) exceptions, or IAx for short — is a typed call-by-name λ-calculus with locally declared ground-type references and a pared down call-by-name version of the simple exceptions (based on ML exceptions) described by Gunther Rémy and Riecke [7]. IAx types are generated from the base types $\mathbf{0}$ (empty), `comm` (commands), `nat` (natural numbers), `var` (natural number references) and `exn` (exceptions).
$T ::= \mathbf{0} \mid \texttt{comm} \mid \texttt{nat} \mid \texttt{var} \mid \texttt{exn} \mid T \Rightarrow T.$
Terms are formed according to the grammar:
$M ::= x \mid \texttt{skip} \mid 0 \mid \texttt{succ}\, M \mid \texttt{pred}\, M \mid \texttt{IF0}\, M \mid$
$\lambda x.M \mid M\, M \mid \mathbf{Y} M \mid M; M \mid$
$\texttt{new_exn}\, M \mid \texttt{mkexn}\, M\, M \mid \texttt{raise}\, M \mid \texttt{handle}\, M\, M \mid$
$\texttt{new}\, M \mid \texttt{mkvar}\, M\, M \mid M := M \mid !M.$
Typing judgements extend those for IA as follows ($B = \mathbf{0} \mid \texttt{comm} \mid \texttt{nat}$):

$$\frac{\Gamma \vdash M : \texttt{exn} \Rightarrow B}{\Gamma \vdash \texttt{new_exn}\, M : B} \qquad \frac{\Gamma \vdash M : \mathbf{0} \Rightarrow \texttt{comm} \quad \Gamma \vdash N : \mathbf{0}}{\Gamma \vdash \texttt{mkexn}\, M\, N : \texttt{exn}}$$

$$\frac{\Gamma \vdash M : \texttt{exn}}{\Gamma \vdash \texttt{raise}\, M : B} \qquad \frac{\Gamma \vdash M : \texttt{exn} \quad \Gamma \vdash N : \mathbf{0}}{\Gamma \vdash \texttt{handle}\, M\, N : \texttt{comm}}$$

The "big step" operational semantics for the imperative fragment of IAx is given in Table 1. Evaluation takes place in an environment consisting of a set of exception names $\mathcal{E}$, a set of variable names or locations $\mathcal{L}$, and a store $\mathcal{S}$ — a partial mapping from $\mathcal{L}$ to natural numbers. By convention, mention of the environment is omitted where possible. The `new_exn` and `new` constants evaluate in the same way; each generates a new name which is added to the environment, and supplied to its argument. Similarly, the `mkvar` construct for generating "bad variables" [15, 1] has a precise analogue in the `mkexn` operation for constructing "bad exceptions"; terms of exception type which may not have the correct raising and handling behaviour.

Programs are evaluated to a final form D, which is either a value V or an exception $E = \texttt{raise}\, h$ for some name h; the latter are propagated through the program until they are *caught*. The handler is simply an operation for capturing a named exception. Because there are no values of type $\mathbf{0}$, $N : \mathbf{0}$ can only evaluate to an exception $\texttt{raise}\, k$, so $\texttt{handle}\, h\, N$ compares the names h and k and evaluates to `skip` if they are equal and propagates the exception $\texttt{raise}\, k$ if they are not. Unlike ML exceptions, in which the use of a universal type of exceptions results in recursive behaviour, the much more restrictive typing of IAx prevents this.

Proposition 2.1 *For any program M of* IAx $- \{\mathbf{Y}\}$, *there is some D such that $M \Downarrow D$.*

A standard notion of observational equivalence can be defined.

Definition 2.2 *Terms $M, N : T$ are* observationally equivalent *(written $M \simeq N$) if for any closing context $C[\cdot] : \texttt{comm}$, $C[M] \Downarrow \texttt{skip}$ if and only if $C[N] \Downarrow \texttt{skip}$.*

Idealized exceptions fit well with the block structure of Idealized Algol and, despite their apparent simplicity, are quite expressive. For instance, although exceptions in Java (and to a lesser extent ML) are more sophisticated in that one handler can be used to trap different exceptions using subtyping, the basic behaviour of Java's `try` and `catch` operations can be captured by defining (for $M, N : \texttt{comm}$, $H : \texttt{exn}$):
$\texttt{try}\, M\, \texttt{catch}\, H\, N =_{df} \texttt{new_exn}\, \lambda k.$
$\texttt{handle}\, k\, ((\texttt{handle}\, H\, (M; \texttt{raise}\, k)); N; \texttt{raise}\, k).$
This executes the command M; if this is completed then the catch block is discarded, but if the exception H is raised whilst running M, then it is caught and the command N is executed.

Exceptions in ML can carry values; this "storage" aspect of exceptions has not been included in IAx because it seems peripheral to the more significant features of exceptions (control-flow manipulation and local declaration) and can be simulated very easily using explicit store; for example, in IAx exceptions carrying natural numbers as values can be represented using $(\texttt{var} \Rightarrow (\texttt{exn} \Rightarrow \texttt{comm})) \Rightarrow \texttt{comm}$ as the type of natural-number-carrying exceptions as follows:
$\texttt{new_exn}\, M =_{df} \texttt{new_exn}\, \lambda x.\texttt{new}\, \lambda y.(M\, \lambda g.(g\, y)\, x)$
$\texttt{raise}\, M\, N : B =_{df} (M\, (\lambda xy.y := N; \texttt{raise}\, x)); \Omega^B$
$\texttt{handle}\, M\, N =_{df}$
$\texttt{new}\, \lambda z.(M\, \lambda xy.(\texttt{handle}\, y\, N); z := !x); !z.$

3 Control Games

The games constructions which will be used to model IAx are based on those given by Hyland and Ong [9] and Nickau [14], in which states of the game are represented as *justified sequences* of moves. Several developments of this basic framework will be used

$$\frac{}{V \Downarrow V} \qquad \frac{L \Downarrow n \quad M \Downarrow \texttt{mkvar}\, N_1\, N_2 \quad N_1\, n \Downarrow D}{M := L \Downarrow D} \qquad \frac{M \Downarrow e}{\texttt{raise}\, M \Downarrow \texttt{raise}\, e}$$

$$\frac{M\, h, \mathcal{L} \cup \{x\} \Downarrow D}{\texttt{new}\, M, \mathcal{L} \Downarrow D}\, x \notin \mathcal{L} \qquad \frac{M \Downarrow \texttt{mkvar}\, N_1\, N_2 \quad N_2 \Downarrow D}{!M \Downarrow D}$$

$$\frac{M \Downarrow V \quad N \Downarrow D}{M; N \Downarrow D} \qquad \frac{N, \mathcal{S} \Downarrow n, \mathcal{S}' \quad M, \mathcal{S}' \Downarrow x, \mathcal{S}''}{M := N, \mathcal{S} \Downarrow \texttt{skip}, \mathcal{S}''[x \mapsto n]} \qquad \frac{M, \mathcal{S} \Downarrow x, \mathcal{S}' \quad \mathcal{S}'(x) = n}{!M, \mathcal{S} \Downarrow n, \mathcal{S}'}$$

$$\frac{M\, h, \mathcal{E} \cup \{h\} \Downarrow D}{\texttt{new_exn}\, M, \mathcal{E} \Downarrow D}\, h \notin \mathcal{E} \qquad \frac{M \Downarrow \texttt{mkexn}\, N_1\, N_2 \quad N_1\, L \Downarrow D}{\texttt{handle}\, M\, L \Downarrow D} \qquad \frac{M \Downarrow \texttt{mkexn}\, N_1\, N_2 \quad N_2 \Downarrow D}{\texttt{raise}\, M \Downarrow D}$$

$$\frac{M \Downarrow E}{\texttt{raise}\, M \Downarrow E} \qquad \frac{M \Downarrow E}{\texttt{handle}\, M\, N \Downarrow E} \qquad \frac{M \Downarrow E}{M; N \Downarrow E}$$

$$\frac{N \Downarrow E}{M := N \Downarrow E} \qquad \frac{M \Downarrow E \quad N \Downarrow n}{M := N \Downarrow E} \qquad \frac{M \Downarrow E}{!M \Downarrow E}$$

$$\frac{M \Downarrow h \quad N \Downarrow \texttt{raise}\, e}{\texttt{handle}\, M\, N \Downarrow \texttt{raise}\, e}\, e \neq h \qquad \frac{M \Downarrow h \quad N \Downarrow \texttt{raise}\, h}{\texttt{handle}\, M\, N \Downarrow \texttt{skip}}$$

Table 1. Operational semantics of exceptions and store

here — in particular the relaxation of constraints to define a model of Idealized Algol [1, 5]. However, it has also been necessary to enrich more significantly the structure on which the games are based — justified sequences — by adding a new notion of 'contingency pointer' to track the flow of control. Fortunately, this fits in relatively smoothly with the original constructions and developments aforementioned.

The structure of a game (the moves, their labels, how they are related) is specified by its *arena*, defined essentially as in [9]. An arena A is a triple:
$\langle M_A, \vdash_A \subseteq (M_A)_* \times M_A, \lambda_A : M_A \to \{Q, A\} \rangle$, where
M_A is a set of tokens called moves,
$\vdash_A \subseteq (M_A)_* \times M_A$ is a relation called *enabling*.
which allows a unique *polarity* for moves to be *inferred* by the following rule — m is an O-move if it is *initial* (i.e. $* \vdash m$), or enabled by a P-move,
m is a P-move if it is enabled by an O-move,
$\lambda_A : M_A \to \{Q, A\}$ is a function which labels moves as *answers* (A) or *questions* (Q), such that every answer has a unique enabling move which is a question.

A *justified sequence* over an arena A is a sequence of elements of M_A in which each occurrence of a non-initial move comes with a *justification pointer* to a preceding occurrence of an enabling move. The transitive closure of justification is referred to as *hereditary justification*. A sequence is *alternating* if Opponent moves are always followed by Player moves, and vice versa.

In order to capture the control behaviour of exceptions in a compositional way, additional pointers of a very similar kind will be added to justified sequences. (The key difference is that there is no structure constraining these pointers analogous to the enabling relation.)

Definition 3.1 *A* contingency pointer *for a move in a justified sequence is a pointer (distinct from its justification pointer) to a preceding* question. *A move is* contingent *if it has such a pointer. A* control sequence *is a justified sequence in which contingency pointers satisfy the same conditions as justification pointers: i.e.*

- *every Player move is contingent on some Opponent move,*
- *every contingent Opponent move is contingent on a Player move,*
- *every answer move is contingent on its enabling question.*

The set of alternating control sequences over the arena A will be written C_A. If a can be reached by following contingency pointers back from c, then c is said to be *hereditarily contingent* on a. To avoid ambiguity caused by multiple occurrences of the same move, we shall sometimes say that in the sequence tb, b is contingent on the prefix $sa \sqsubseteq tb$ instead of saying that b is contingent on a.

3.1 A Category

A category of arenas and strategies can now be defined using the standard constructions [9, 12].

Product For any set-indexed family of arenas $\{A_i \mid i \in I\}$, form the product $A = \Pi_{i\in I} A_i$ as follows:

- $M_{\Pi_{i\in I} A_i} = \coprod_{i\in I} M_{A_i}$,
- $\langle m, i\rangle \vdash_{\Pi_{i\in I} A_i} \langle n, j\rangle$ if $i = j$ and $m \vdash_{A_i} n$, and $* \vdash_{\Pi_{i\in I} A_i} \langle n, j\rangle$ if $* \vdash_{A_j} n$,
- $\lambda^{QA}_{\Pi_{i\in I} A_i}(\langle m, i\rangle) = \lambda_{A_i}(m)$.

For finite k, the product of k copies of the arena A will be written A^k.

Function Space For arenas A_1, A_2

- $M_{A_1 \Rightarrow A_2} = M_{A_1} + M_{A_2}$,
- $\langle m, i\rangle \vdash_{A\Rightarrow B} \langle n, j\rangle$ if $i = j$ and $m \vdash n$ *or* $m \in M_B$, $n \in M_A$, $* \vdash_B m$ and $* \vdash_B n$, $* \vdash \langle m, i\rangle$ if $m \in M_B$ and $* \vdash_B m$,
- $\lambda^{QA}_{A\Rightarrow B}(\langle m, i\rangle) = \lambda_{A\Rightarrow B}(m)$.

The arena with a single question move is written o.

Definition 3.2 *A (deterministic) strategy over an arena A is a non-empty even-prefix-closed set of even length alternating justified sequences which is evenly branching: $sa, sb \in \sigma \implies b = c$.*
A control-strategy *on A is a strategy consisting of control-sequences (i.e. a subset of C_A).*

The control-strategies will be referred to simply as strategies where the context is clear.

Composition of control-strategies is a straightforward extension of 'parallel composition with hiding' [6] to control sequences.
If $s \in C_{A_1\Rightarrow(A_2\Rightarrow A_3)}$ then $s\restriction(A_i, A_j)$ is a sequence with contingency pointers (not necessarily a true control sequence) defined as follows:
$\epsilon\restriction(A_i, A_j) = \epsilon$,
$sa\restriction(A_i, A_j) = s\restriction(A_i, A_j)$ if $a \notin A_i, A_j$,
$sa\restriction(A_i, A_j) = (s\restriction(A_i, A_j))a$ if $a \in A_i, A_j$,
where a is justified by the most recently played move from A_i or A_j which hereditarily justifies a in s (if any) and a is contingent on the most recent move from A_i or A_j on which it is hereditarily contingent.

Definition 3.3 *For $\sigma : A_1 \to A_2, \tau : A_2 \to A_3$*
$\sigma;\tau = \{s \in C_{A_1,A_3} \mid \exists t \in C_{((A_1\Rightarrow A_2)\Rightarrow A_3)} \cdot$
$t\restriction(A_1, A_2) \in \sigma \wedge t\restriction(A_2, A_3) \in \tau \wedge t\restriction A_1, A_3 = s\}$.

As usual, canonical morphisms are *copycat* strategies which just copy Opponent moves between different parts of a game. However, contingency pointers (unlike justification pointers) are *not* copied; to define copycat control-strategies requires the notion of *pending question.*

Definition 3.4 *Define the "pending question prefix" of a justified sequence as follows:*
$\mathsf{pending}(\varepsilon) = \varepsilon$,
$\mathsf{pending}(sa) = sa$, *if a is a question,*
$\mathsf{pending}(satb) = \mathsf{pending}(s)$, *if b is an answer to a.*

Definition 3.5 *For any arena A, define the identity control-strategy $\mathsf{id}_A : A \Rightarrow A$ to be the least subset of $C_{A\Rightarrow A}$ containing ε and closed under the condition:*
If $s \in \mathsf{id}_A$ and $sab\restriction A^+ = s\restriction A^-$ and b is contingent on (the last move in) $\mathsf{pending}(sa)$ *then $sab \in \mathsf{id}_A$.*

So, for example, in the play of $\mathsf{id}_{o\Rightarrow o}$ represented in Figure 1, the last move is contingent on its immediate predecessor, but justified by the initial move. As

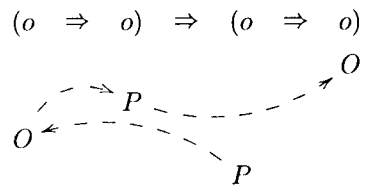


Figure 1. A play of $\mathsf{id}_{o\Rightarrow o}$ (with contingency pointers)

for general strategies [12], arenas and control-strategies form a SMCC which can be refined to a CCC of *well-opened strategies.*

Definition 3.6 *The* thread *of the last move in a non-empty control sequence is defined as follows:*
$\mathsf{thread}(sa) = a$, *(a initial)*
$\mathsf{thread}(satb) = \mathsf{thread}(sa)b$. *(a is the last move in sat justified by the same initial move as b).*
b is contingent on the most recent move in $\mathsf{thread}(sa)$ *on which it is hereditarily contingent in satb.*
A strategy σ is well-opened *if every control sequence in σ contains at most one initial move.*
If $\tau : A$ is a well-opened strategy, then $\tau^\dagger : A$ is the least subset of C_A containing ε and closed under the condition:
if $s \in \tau^\dagger$, and $\mathsf{thread}(sab) \in \tau$, *then $sab \in \tau^\dagger$.*

The well-opened identity is the restriction of id_A to well-opened sequences.

Thus we have two cartesian closed categories of games, both of which have arenas as objects and well-opened strategies over the function-space $A \Rightarrow B$ as morphisms from A to B, with composition defined $\sigma \cdot \tau = \tau^\dagger;\sigma$:

$\mathcal{G}, \mathbf{1}, \times, \Rightarrow$ — the category of games, which has (general) strategies as morphisms — and $\mathcal{CG}, \mathbf{1}, \times, \Rightarrow$ — the category of *control games* which has control-strategies as morphisms.

Apart from exception-declaration and handling, the semantics of IAx is given by an embedding which takes the semantics of IA in $\mathcal{G}$ [1] to $\mathcal{CG}$. To define this embedding requires the notion of *well-bracketing.*

Definition 3.7 *A strategy σ in $\mathcal{G}$ is well-bracketed if every* answer *played by σ is justified by the pending question. A control-strategy σ in $\mathcal{CG}$ is* well-bracketed *if every* move *made by σ is contingent on the pending question: i.e. if $sa \in \sigma$, then b is contingent on* pending(sa).

The well-bracketed strategies form cartesian closed subcategories of $\mathcal{G}$ and $\mathcal{CG}$, which will be written $\mathcal{G}_{WB}$ and $\mathcal{CG}_{WB}$. All of the strategies required to interpret IA [1] are well-bracketed.

Definition 3.8 *For any control sequence s, let $|s|$ be the underlying justified sequence obtained by forgetting the contingency pointers.*
For a control-strategy σ, let $|\sigma| = \{|s| \ : \ s \in \sigma\}$. Say that σ is control-blind *if $|\sigma|$ is a deterministic strategy.*

Proposition 3.9 *There is an embedding of $\mathcal{G}_{WB}$ into $\mathcal{CG}$, which has as its image the well-bracketed and control-blind strategies.*

PROOF: For any $\sigma : A \in \mathcal{G}_{WB}$ define $\widetilde{\sigma}$ to be the least subset of C_A containing ε and closed under the following condition:
If $s \in \widetilde{\sigma}$, and $|sab| \in \sigma$ and b is contingent on pending(sa) then $sab \in \sigma$.
Then $\widetilde{\sigma}$ is a well-bracketed strategy (well-bracketedness of σ implies that every answer is contingent on its justifying question) and $\widetilde{(_)}$ is compositional and preserves cartesian closed structure. For any $\tau \in \mathcal{G}^{WB}$, $|\widetilde{\tau}| = \tau$, and for any well-bracketed and control-blind $\sigma \in \mathcal{CG}$, $\widetilde{|\sigma|} = \sigma$. □

4 Semantics of Exceptions

The interpretation of locally bound exceptions given here is based on viewing elements of exception type h : exn as 'objects' defined by their 'methods' — in this case raise h : comm and handle h : comm $\Rightarrow$ comm. This was suggested as an interpretation for *reference types* by Reynolds [15] and followed in a game semantics setting in [1, 5].

The type exn is interpreted as the arena exn $= (\llbracket \mathbf{0} \rrbracket \Rightarrow \llbracket \mathtt{comm} \rrbracket) \times \llbracket \mathbf{0} \rrbracket$ (where $\llbracket \mathtt{comm} \rrbracket$ is the arena with one question and one answer, and $\llbracket \mathbf{0} \rrbracket$ is the arena o with just a question). The initial questions in the two components $\llbracket \mathbf{0} \rrbracket \Rightarrow \llbracket \mathtt{comm} \rrbracket$ and $\llbracket \mathbf{0} \rrbracket$ will be referred to as handle and raise respectively. The answer to handle will be referred to as caught, and the question enabled by handle as ok. The handle and raise methods are the first and second projections from exn; mkexn is pairing.

$$\llbracket \Gamma \vdash \mathtt{handle}\, M\, N \rrbracket = \langle \llbracket \Gamma \vdash M \rrbracket; \pi_l, \llbracket \Gamma \vdash N \rrbracket \rangle; \mathsf{App}$$

$$\llbracket \Gamma \vdash \mathtt{raise}\, M : B \rrbracket = \llbracket \Gamma \vdash M \rrbracket; \pi_r; \mathsf{Wk}_{\llbracket B \rrbracket}$$

$$\llbracket \Gamma \vdash \mathtt{mkexn}\, M\, N \rrbracket = \langle \llbracket \Gamma \vdash M \rrbracket, \llbracket \Gamma \vdash N \rrbracket \rangle$$

(Where $\mathsf{Wk}_A : o \Rightarrow A$ is the strategy which responds to the initial question in A with the unique question in o.) Thus the only part of IAx which is *not* represented by a control-blind and well-bracketed strategy is new-exception declaration. This is defined using composition with a strategy xcell (similar to the strategy cell which gives the denotation of new [1]) that uses contingency pointers in an essential way to match up raises and handles appropriately, via the notion of an *open question.*

Definition 4.1 *The set of prefixes of a control sequence which* terminate in an open question *is defined by induction on length, as follows:*
open$(\varepsilon) = \{\}$,
open$(sa) = \{sa\}$, *if a is not contingent,*
if b is contingent on a, then:
open$(satb)$ = open(s) *if* $\lambda^{QA}(b) = A$,
open$(satb)$ = open$(sa) \cup \{sa \cdot tb\}$, *otherwise.*

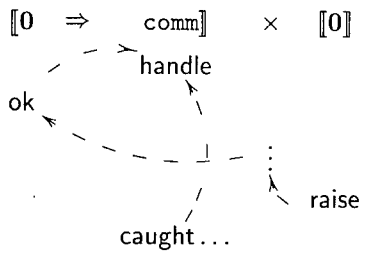


Figure 2. A typical play of xcell

A "typical play" of xcell is depicted in Figure 2 (arrows are contingency pointers). Its behaviour can be described informally as follows.

- If Opponent plays a handle move then xcell responds with an 'ok' move, justified by (and contingent on) it.

- If Opponent plays a raise move and some handle moves are open, then xcell answers the most recently played one. If there is no open handle question, then xcell does nothing — this represents divergence caused by an uncaught exception.

Definition 4.2 *Let the strategy* xcell : exn *be the least subset of* C_{exn} *containing* ε *and closed under the following conditions:*
if $s \in \mathsf{xcell}$, *then* $s \cdot \mathsf{handle} \cdot \mathsf{ok} \in \mathsf{xcell}$ *(where* ok *is contingent on* $s \cdot \mathsf{handle}$*),*
if $t \in \mathsf{xcell}$, *and* $s \cdot \mathsf{handle} \in \mathsf{open}(t \cdot \mathsf{raise})$, *and for all* $r \cdot \mathsf{handle} \in \mathsf{open}(t \cdot \mathsf{raise})$, $r \sqsubseteq s$, *then* $t \cdot \mathsf{raise} \cdot \mathsf{caught} \in \mathsf{xcell}$, *where* caught *is contingent on* $s \cdot \mathsf{handle}$.

$$[\![\Gamma \vdash \mathtt{new_exn}\, M]\!] = ([\![\Gamma \vdash M]\!] \times \mathsf{xcell}); \mathsf{App}.$$

4.1 Soundness

Soundness of the interpretation with respect to the operational semantics can now be established; the only novel feature of the proof is that it requires meanings to be assigned to programs which raise exceptions.
Given M : comm or M : nat, $\mathcal{E} = e_1, \ldots e_n$, $\mathcal{L} = x_1, \ldots x_m$, and $k \leq m$ such that $\mathcal{S}(x_i) \downarrow$ if and only if $i \leq k$, let $\mathtt{new}\, \mathcal{L} := \mathcal{S}\ \mathtt{in}\ M =_{df}$
$\mathtt{new}\, \lambda x_1 \ldots \mathtt{new}\, \lambda x_m . x_1 := \mathcal{S}(x_1); \ldots; x_k := \mathcal{S}(x_k); M$.
Then $[\![M, \mathcal{E}, \mathcal{L}, \mathcal{S}]\!]$ is the *unique* maximal-length sequence in $[\![e_1, \ldots, e_n \vdash \mathtt{new}\, \mathcal{L} := \mathcal{S}\ \mathtt{in}\ M]\!]$ such that $s \restriction \mathsf{exn}^n \in \mathsf{xcell}^n$.
Soundness is proved (by induction on derivation, using standard facts about the model together with analysis of xcell) with respect to the following binary approximation relation ($\sim$):
$[\![M, \mathcal{E}, \mathcal{L}, \mathcal{S}]\!] \sim [\![M', \mathcal{E}', \mathcal{L}', \mathcal{S}']\!]$ if the last move in $[\![M, \mathcal{E}, \mathcal{L}, \mathcal{S}]\!]$ is the same as the last move in $[\![M', \mathcal{E}', \mathcal{L}', \mathcal{S}']\!]$.

Proposition 4.3 *If* $M, \mathcal{E}, \mathcal{L}, \mathcal{S} \Downarrow D, \mathcal{E}', \mathcal{L}', \mathcal{S}'$ *then* $[\![M, \mathcal{E}, \mathcal{L}, \mathcal{S}]\!] \sim [\![D, \mathcal{E}', \mathcal{L}', \mathcal{S}']\!]$.

The interpretation is also *adequate*. This follows directly from soundness and termination of all evaluations of $\mathbf{Y}$-free terms (Proposition 2.1).

Proposition 4.4 *For any* IAx *program* M : comm, $[\![M]\!] \neq \bot$ *if and only if* $M \Downarrow \mathtt{skip}$.

PROOF: The proof of completeness is by induction on the number of occurrences of $\mathbf{Y}$ in M. Suppose $[\![M]\!] \neq \bot$. By proposition 2.1 $M \Downarrow D$ for some D, and $D = \mathtt{skip}$ by soundness. If $M = C[\mathbf{Y}N]$ for some $\mathbf{Y}$-free N, then $C[\mathbf{Y}N] = \bigsqcup_{i \in \omega} [\![C[N^i]]\!] \neq \bot$ (where $N^0 = \Omega$, and $N^{k+1} = N\ N_k$). Hence $C[N^k] \neq \bot$ for some $k \in \omega$ and by induction $C[N^k] \Downarrow$ and an induction on derivations shows that $C[\mathbf{Y}N] \Downarrow$. □

5 A Fully Abstract Model

An adequate model of IAx with exceptions has been described which is not fully abstract because it lacks the following 'definability property'.

Definition 5.1 *A model* $\mathcal{M}$ *of* IAx *has the definability property if for every context* Γ *and type* T, *every (compact)* $f : [\![\Gamma]\!] \to [\![T]\!]$ *in* $\mathcal{M}$ *is definable; i.e. there exists an* IAx *term* M_f *such that* $f = [\![\Gamma \vdash M_f : T]\!]$.

In this section, the category of control games will be cut down so that all compact strategies are definable in IAx by giving a series of semantic *definability criteria*, and hence a full abstraction result will be achieved. The criteria are based on constraining three aspects of behaviour on control games; which moves Player's *contingency pointers* can point to (a variant of the bracketing condition), which moves Player's *justification pointers* can point to (a variant of the *visibility condition* [9]) and a new condition governing which of Opponent's contingency pointers can be observed by Player.

Definition 5.2 (Weak Bracketing) *A strategy* σ *is* weakly bracketed *if every Player move in* σ *is contingent on an open question — i.e. if* $sb \in \sigma$ *where* b *is contingent on* $ta \sqsubseteq sb$ *then* $ta \in \mathsf{open}(s)$.

The notion of *view*, defined for justified sequences in [9], extends to control sequences in line with the intuition that when Player makes a move contingent on an earlier move it may be regarded as if they ocurred in direct succession.

Definition 5.3 (View) *The Player-view of a control sequence is defined as follows:*
$\ulcorner sa \urcorner = a$, *if* a *is initial.*
$\ulcorner satb \urcorner = \ulcorner sa \urcorner b$ *if* b *is an* O*-move* justified by a,
$\ulcorner satb \urcorner = \ulcorner sa \urcorner b$ *if* b *is a* P*-question* contingent on a,
$\ulcorner satb \urcorner = \ulcorner s \urcorner$ *if* b *is a* P*-answer to* a.

This accords with the original notion of views given in [9] in that for any well-bracketed strategy $\sigma \in \mathcal{CG}$, $s \in \sigma$ implies that $\ulcorner s \urcorner = \ulcorner |s| \urcorner$. It "dualizes" to a notion of O-view ($\llcorner _ \lrcorner$) as in [9].

Definition 5.4 (Visibility) *A strategy (in* $\mathcal{CG}$ *or* $\mathcal{G}$*) satisfies the* visibility condition *if for every* $s \in \sigma$, $\ulcorner s \urcorner$ *is a well-defined* justified sequence. *The cartesian closed subcategory* $\mathcal{G}$ *of well-bracketed strategies satisfying visibility will be written* $\mathcal{G}_{V \cdot B}$.

A (well-bracketed) strategy σ satisfies visibility if and only if $|\sigma|$ satisfies visibility. Hence the embedding of $\mathcal{G}_{W \cdot B}$ into $\mathcal{CG}$ restricts to $G_{V \cdot B}$.

The third definability criterion limits the power of Player to observe contingency pointers. (It corresponds

to the fact that in IAx the only way to observe exception handling is by raising and handling a competing exception.)
Let $satb$ be a control sequence in which b is a P-move contingent on a, and let $rc \sqsubseteq satb$ where c is a O-move. Then c is *prematurely closed* by b if $rc \in \mathsf{open}(sat)$ and $rc \notin \mathsf{open}(sa)$. Player's *perspective* on a control sequence is obtained by deleting the contingency pointers which are not attached to O-questions prematurely closed by some P move. It can be defined concisely (for extensions to control-sequences) as follows:
$\lceil \varepsilon \rceil = \varepsilon$,
If b is contingent on a, then $\lceil satb \rceil = \lceil s \rceil a \lceil t \rceil b$, where the pointer from b to a is included if and only if a is a P-question, and all of the pointers from $a\lceil t \rceil$ into s are omitted.

Definition 5.5 *A strategy is control-innocent if whenever $sab, t \in \sigma$ and $\lceil sab \rceil = \lceil tab \rceil$, then $tab \in \sigma$.*

Proposition 5.6 *If σ is well-bracketed and control-innocent then σ is control-blind.*

PROOF: If σ is a well-bracketed strategy, then $\lceil s \rceil = |s|$ as σ closes only pending O-questions. □

Hence the image of the embedding of $\mathcal{G}_{VB}$ into $\mathcal{CG}$ consists of the well-bracketed strategies satisfying visibility and control-innocence. The following proposition is just a straightforard extension of the definability theorem for IA [1] to include the base type exn.

Proposition 5.7 *All finite strategies in $\mathcal{G}_{VB}$ over* IAx − {new_exn} *type-objects are definable in* IAx − {new_exn}.

Corollary 5.8 *The (compact) definable strategies of* IAx − {new_exn} *are the well-bracketed and control-innocent finite strategies which satisfy visibility.*

5.1 Factorization and Definability

The finite, weakly-bracketed, visibility-satisfying and control-innocent strategies can now be identified as the compact IAx-definable morphisms by showing that they are obtained by composing xcell with the well-bracketed and and control blind strategies.

Definition 5.9 *Define $\mathcal{CG}_{/\mathsf{xcell}}$ to be the cartesian closed subcategory of control games in which morphisms are* finite *strategies $f : A \to B$ such that there exists $k \in \omega$ and a well-bracketed strategy $g : A \times \mathsf{exn}^k \to B$ such that $\mathsf{id} \times \mathsf{xcell}^k; g = f$.*

Proposition 5.10 *The compact elements of $\mathcal{CG}$ which are definable in* IAx *are precisely the morphisms of $\mathcal{CG}_{/\mathsf{xcell}}$.*

PROOF: It is straightforward to establish by structural induction on M that every $[\![\Gamma \vdash M : T]\!]$ is the least upper bound of a chain of approximants in $\mathcal{CG}_{/\mathsf{xcell}}$.
Conversely, if $\sigma : [\![\Gamma]\!] \to [\![T]\!]$ is a morphism in $\mathcal{CG}_{/\mathsf{xcell}}$ then there is a well-bracketed — and hence IAx − {new_exn} definable — strategy $\widehat{\sigma} : [\![\Gamma]\!] \times \mathsf{exn}^k \to (A \Rightarrow B)$ such that $\sigma = \mathsf{id}_{[\![\Gamma]\!]} \times \mathsf{xcell}^k; \widehat{\sigma}$, and hence $\sigma = [\![\Gamma \vdash \mathtt{new_exn}\, \lambda x_1 \ldots \mathtt{new_exn}\, \lambda x_k . M_\sigma]\!]$. □

Proposition 5.11 *A strategy is in $\mathcal{CG}_{/\mathsf{xcell}}$ if and only if it is finite, weakly-bracketed, control-innocent and satisfies visibility.*

Proof of this proposition comes in two parts; first it is shown that if $\sigma : \mathsf{exn} \to A$ satisfies weak-bracketing, control-innocence and visibility then so does $\mathsf{xcell}; \sigma$, which is a consequence of the following two lemmas.

Lemma 5.12 *Suppose $\sigma : \mathsf{exn} \Rightarrow A$ and $sa \in \sigma$ is such that a is a move in A, and $s\lceil \mathsf{exn} \in \mathsf{xcell}$. Then $\mathsf{open}(sa)\lceil M_A = \mathsf{open}(sa\lceil A)$ and $\ulcorner sa \urcorner \lceil A = \ulcorner sa \lceil A \urcorner$.*

Lemma 5.13 *Suppose $\sigma : \mathsf{exn} \Rightarrow A$ is control-innocent, and $sab, t \in \sigma$ where $s \lceil \mathsf{exn}, t \lceil \mathsf{exn} \in \mathsf{xcell}$ and b is a move in A such that $\lceil sab \lceil A \rceil = \lceil tab \lceil A \rceil$. Then $tab \in \sigma$.*

The second part of the proof of Proposition 5.11 is to show that all weakly-bracketed strategies can be obtained from well-bracketed strategies by composition with xcell (so in fact the stronger result that every compact strategy can be defined using a single exception variable is established). This is achieved by methods similar to the factorizations described in [1, 10, 5, 8], in this case using the jump in control between the raise and caught moves of xcell to generate all of the control jumps in a weakly-bracketed strategy. The complicating factor is that the properties of control-innocence and visibility must be maintained. In particular, forcing Opponent to close questions instead of Player hides their contingency pointers — as has already been observed, well-bracketed strategies cannot observe any pointers at all. So it is necessary to make all of the information carried by the perspectives of σ manifest as explicit exception handling.

The factorization of a strategy $\sigma : B$ to $\widehat{\sigma} : \mathsf{exn} \to B$ by adding handle, ok and raise, caught moves in exn (see Figure 3), can be informally described as follows.

- Immediately before playing a *question* in A, $\widehat{\sigma}$ plays a handle move (contingent on the pending question), to which Opponent responds with ok.
- If σ responds to sa by playing a move b which prematurely closes n Opponent moves then $\widehat{\sigma}$ responds to $\widehat{sa}$ by playing n raise moves — each of

which is caught by a handler corresponding to one of the O-moves which are closed by b — until all of these O-moves have been closed. If b is an answer then $\widehat{\sigma}$ plays b contingent on the pending question; if b is a question, then $\widehat{\sigma}$ plays a handle (as above) and then plays b pointing to the pending question.

Note that as all of the contingency pointers from the control-view of σ are used to match up the raises and handles, they are now observable as play in the premiss exn.

Proposition 5.14 (Control Factorization) *If $\sigma : A$ is a finite, weakly-bracketed and control-innocent strategy (satisfying visibility) then there is some finite, well-bracketed strategy $\widehat{\sigma} : \mathsf{exn} \to A$ (satisfying visibility) such that* $\mathsf{xcell}; \widehat{\sigma} = \sigma$.

PROOF: is by defining $\widehat{\sigma} = \{t \sqsubseteq^{even} \widehat{s} \mid s \in \sigma\}$, where $\widehat{(_)} : C_A \to C_{\mathsf{exn} \Rightarrow A}$ is a translation on even-length control sequences such that:

- $\widehat{s}\restriction A = s$ and $\widehat{s}\restriction \mathsf{exn} \in \mathsf{xcell}$,
- every Player move in $\widehat{s}$ is contingent on the pending question,
- if $\ulcorner s \urcorner$ is well-defined then so is $\ulcorner \widehat{s} \urcorner$,
- $\lceil s \rceil = \lceil t \rceil$ if and only if $\widehat{s} = \widehat{t}$.

For an even-length sequence s, let $\psi(s)$ be the number of O-questions prematurely closed by the last move in s. Now define $\widehat{s}$ by induction on sequence length:
$\widehat{\varepsilon} = \varepsilon$,
$\widehat{spq} = \widehat{s}p(\mathsf{raise}\,\mathsf{caught})^{\psi(spq)}(\mathsf{handle}\,\mathsf{ok})q$,
$\widehat{spa} = \widehat{s}p(\mathsf{raise}\,\mathsf{caught})^{\psi(spa)}a$,
where $\lambda^{QA}(q) = Q$, and $\lambda^{QA}(a) = A$. □

5.2 Full Abstraction

In a now-standard fashion, definability for the compact elements of the model of IAx yields full abstraction for its "intrinsic preorder collapse". Moreover, this fully abstract model can be described directly, showing that it is effectively presentable.

Definition 5.15 *Given strategies $\sigma, \tau : A$, $\sigma \lesssim_A \tau$ if for every well-bracketed strategy $\rho : A \to [\![\mathsf{comm}]\!]$, $\sigma; \rho \neq \{\varepsilon\}$ implies $\tau; \rho \neq \{\varepsilon\}$.*
$\sigma \equiv \tau$ if $\sigma \lesssim_A \tau$ and $\tau \lesssim_A \sigma$.

Theorem 5.16 (Full Abstraction) *For any closed terms $M, N : T$, $[\![M]\!]_{\mathcal{CG}} \equiv [\![N]\!]_{\mathcal{CG}}$ if and only if $M \simeq N$.*

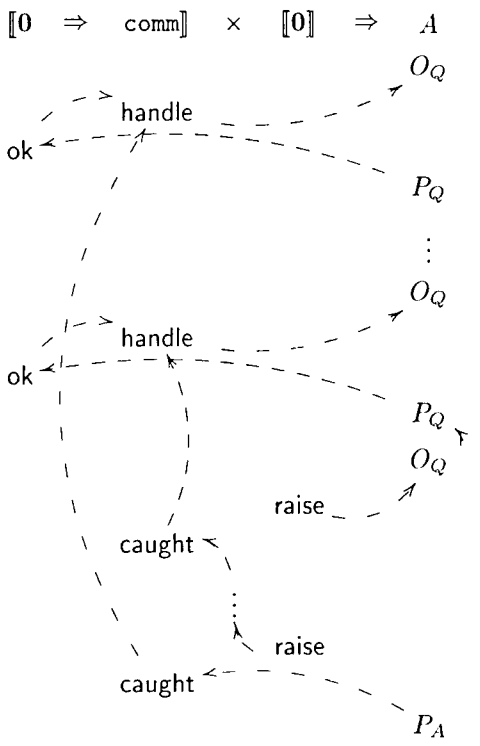


Figure 3. Factorization of a control jump

In fact, the equationally fully abstract model can be directly presented simply by including visibility and bracketing in the definition of a legal play.

Definition 5.17 *An alternating control sequence s over an arena A is* legal *if both Player and Opponent satisfy the weak-bracketing and visibility conditions: i.e.*
$sa \in L_A$ if and only if $s \in L_A$ and if a is contingent on b then $b \in \mathsf{open}(s)$, and $\ulcorner sa \urcorner$ and $\llcorner sa \lrcorner$ are both well-formed justified sequences.
For $\sigma : A$, write $L(\sigma)$ for $\sigma \cap L_A$.

Lemma 5.18 *For any $\sigma, \tau : A$, $L(\sigma) \subseteq L(\tau)$ if and only if $\sigma \lesssim_A \tau$.*

PROOF: To prove the implication from right to left (showing that control-innocence does not affect the intrinsic preorder) suppose $L(\sigma) \not\subseteq L(\tau)$. Let $sb \in \sigma$ be a minimal-length control sequence such that $sb \notin \tau$. Let q be the initial question in comm and a its answer), and define:

$$\rho : A \to [\![\mathsf{comm}]\!] = \{t \in C_{A \Rightarrow [\![\mathsf{comm}]\!]} \mid |\lceil t \rceil| \sqsubseteq^{even} \lceil qsba \rceil\}$$

Then ρ is by definition a control-innocent strategy such that $\rho; \sigma \neq \{\varepsilon\}$. Moreover, $\rho; \tau = \{\varepsilon\}$ — if $sc \in \rho$ then

either $|sc| \neq |sb|$ or c and b are contingent on different moves, and hence $\lceil qsca \rceil \neq \lceil qsba \rceil$. □

6 Contingency and Expressiveness

The complex structure of control games provokes the question: Are contingency pointers really necessary to model exceptions? The category $\mathcal{G}$ contains models of a wide range of sequential features, including both references [1, 5] and **call/cc** [10, 11] at all types. Might there not be a semantics of exceptions in $\mathcal{G}$? Part of the interest in this question arises because it is closely related to a problem which is both independent of semantics, and an area of current research interest: When can one combination of programming language features be macro-expressed in terms of another [17, 18]? For example, it is "folklore" [18] that exceptions may be expressed in terms of continuations and references. As both of the latter can be modelled in $\mathcal{G}$, if the folklore were true then a semantics of exceptions in $\mathcal{G}$ could be given by factoring through this interpretation. On the other hand, given a model of exceptions, continuations and references in $\mathcal{G}$ it should be possible to use the a combination of the definability results for references and continuations to extract an encoding of exceptions.

In fact, it is not possible to give a semantics of exceptions which is a conservative extension of the model of IA in $\mathcal{G}$, and hence it is *not* possible to macro-express exceptions using continuations and references. A paper is in preparation which contains formal proofs of the latter claim, using syntactic counterexamples extracted from the game semantics of exceptions, continuations and references. This section will sketch a proof of the former claim, showing how differences in contingency structure can cause differences in observable behaviour. A starting point is the observation there are strategies which contain the same underlying justified sequences, but are observationally distinct because they have different contingency pointers. A simple arena in which this may be observed is $(o \Rightarrow o) \Rightarrow (o \Rightarrow o))$ which will be called A_1 for short; it is the denotation of the type $T_1 = (\mathbf{0} \Rightarrow \mathbf{0}) \Rightarrow (\mathbf{0} \Rightarrow \mathbf{0})$. Recall that in the identity strategy on $o \Rightarrow o$ (Figure 1) Player moves are always contingent on the preceding O-move.

Proposition 6.1 *There is a weakly bracketed, visibility-satisfying and control-innocent strategy* $\mathsf{not_id} : A_1$ *such that* $|\mathsf{not_id}| = |\mathsf{id}_{o \Rightarrow o}|$ *but* $\mathsf{id} \not\equiv \mathsf{not_id}$.

PROOF: Let not_id be the strategy consisting of the even prefixes of the play depicted in Figure 4. Then $|\mathsf{not_id}_{o \Rightarrow o}| = |\mathsf{id}_{o \Rightarrow o}|$ but $\mathsf{not_id} \not\equiv \mathsf{id}$ by Lemma 5.18. □

Moreover (as the definability and full abstraction results imply) not_id and id are the denotations of terms which are not observationally equivalent:
$\mathsf{id} = [\![\lambda f.f]\!]$, $\mathsf{not_id} = [\![\mathrm{NOT_ID}]\!]$ where $\mathrm{NOT_ID} = \lambda f.\lambda x.\mathtt{new_exn}\,\lambda h.(\mathtt{handle}\,h\,(f\,\mathtt{raise}\,h)); x$.
$\mathrm{NOT_ID} \not\equiv \lambda f.f$; let $\mathrm{ID_TEST} : T_1 \Rightarrow \mathtt{comm} = \lambda g.\mathtt{new_exn}\,k.\mathtt{handle}\,k\,(((g\,\lambda x.\mathtt{handle}\,k\,x)\,\mathtt{raise}\,k); \Omega)$:
$\mathrm{ID_TEST\ NOT_ID} \Downarrow \mathtt{skip}$ and $\mathrm{ID_TEST}\ \lambda f.f \not\Downarrow \mathtt{skip}$.

The distinction between not_id and id can be used to show that there is no model of IAx in $\mathcal{G}$.

Definition 6.2 *Define the $\mathcal{G}$-strategy* $\mathsf{idtrunc} : A_1 \to A_1 = \{t \in L_{A_1^- \Rightarrow A_1^+} \mid t \in \mathsf{id}_{A_1} \wedge t\lceil A_1^- = t\lceil A_1^+ \in \mathsf{id}_{o \Rightarrow o}\}$

As the definability result of [1] entails, idtrunc is definable as a term of Idealized Algol:
$\mathrm{TRUNC} = \lambda g : T.\mathtt{new}\,\lambda z.\lambda f.\lambda x.z := 0; ((g\,M_1)\,M_2)$, where $M_1 = \lambda y.\mathrm{IF0}\ !z\ \mathrm{then}\ (z := 1; (f\,y))\ \mathrm{else}\ \Omega)$ and $M_2 = \mathrm{IF0}\ !z\ \mathrm{then}\ \Omega\ \mathrm{else}\ x$.

Lemma 6.3 *For all $\sigma : A_1$ in $\mathcal{G}$, $\sigma; \mathsf{idtrunc} \lesssim_{A_1} \mathsf{id}_{o \Rightarrow o}$*

PROOF: This is direct by definition of idtrunc. □

In $\mathcal{CG}$, $\mathsf{not_id}; \widetilde{\mathsf{idtrunc}} = \mathsf{not_id} \not\lesssim_{A_1} \mathsf{id}_{o \Rightarrow o}$ — and this fact can be exploited to prove the following.

Proposition 6.4 *There is no adequate model of IAx in $\mathcal{G}$ which conservatively extends the semantics of IA.*

PROOF: Suppose there is such an interpretation. Then $\mathrm{ID_TEST\ (TRUNC\ NOT_ID)} \Downarrow \mathtt{skip}$ implies that $[\![\mathrm{ID_TEST\ (TRUNC\ NOT_ID)}]\!]_{\mathcal{G}} \neq \bot$, and hence $([\![\mathrm{NOT_ID}]\!]_{\mathcal{G}}; \mathsf{idtrunc}); [\![\mathrm{ID_TEST}]\!]_{\mathcal{G}} \neq \bot$. By Lemma 6.3, $[\![\lambda f.f]\!]_{\mathcal{G}}; [\![\mathrm{ID_TEST}]\!]_{\mathcal{G}} \neq \bot$, and so $[\![\mathrm{ID_TEST}\ \lambda f.f]\!]_{\mathcal{G}} \neq \bot$. But this contradicts adequacy, as $\mathrm{ID_TEST}\ \lambda f.f \not\Downarrow \mathtt{skip}$. □

6.1 Further Directions

By demonstrating that the games semantics of exceptions requires new structure, unlike the models of

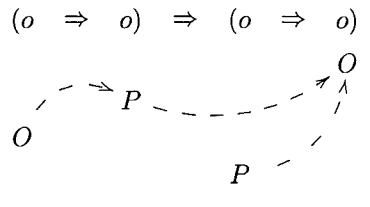


Figure 4. A typical play of not_id

references and continuations, we have shown the limitations of the "semantic cube" [4] of models of programming language features based simply upon relaxing constraints on the original model of PCF. But the basic analyis implicit in the cube is strengthened by the new structure — in effect, we have added an extra dimension to it. The extra degree of freedom available in the category of control games can be exploited to give a thorough analysis of the interactions between exceptions, continuations and references. The latter can be modelled by dropping the "visibility condition" in the style of [1] to reach a fully abstract semantics of "core ML". (It is straightforward to move to a call-by-value perspective by using, for instance, the $\mathsf{Fam}(\mathcal{C})$ construction [2].)

To allow **call/cc** to be interpreted, the weak bracketing condition is relaxed. In this model, throwing a continuation and and handling an exception both correspond to playing a move which is not contingent on the pending question; the distinguishing feature of exceptions is that they allow contingency pointers to be *observed*. The most interesting feature of the model is that (unlike the model of continuations in $\mathcal{G}$ [11]) it is not an example of continuation-passing-style construction; it contains observably distinct strategies which represent terms which are equivalent in all cps models. These terms constitute a further counterexample to the claim that exceptions can be expressed using continuations, which can be presented wihtout recourse to the game semantics.

References

[1] S. Abramsky and G. McCusker. Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions. In P. O'Hearn and R. Tennent, editors, *Algol-like languages*. Birkhauser, 1997.

[2] S. Abramsky and G. McCusker. Call-by-value games. In M. Neilsen and W. Thomas, editors, *Computer Science Logic: 11^{th} Annual workshop proceedings*, LNCS, pages 1–17. Springer-Verlag, 1998.

[3] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. To appear in Theoretical Computer Science, 1998.

[4] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenburg and U. Berger, editors, *Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School*. Springer-Verlag, 1998.

[5] S. Abramsky, K. Honda, G. McCusker. A fully abstract games semantics for general references. In *Proceedings of the 13th Annual Symposium on Logic In Computer Science, LICS '98*, 1998.

[6] S. Abramsky, R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59:543–574, 1994.

[7] C. Gunter, D. Remy, and J. Riecke. A generalization of exceptions and control in ML like languages. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, 1995.

[8] R. Harmer and G. McCusker. A fully abstract games semantics for finite non-determinism. In *Proceedings of the Fourteenth Annual Symposium on Logic in Computer Science, LICS '99*. IEEE press, 1998.

[9] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III, 1995. To appear in Theoretical Computer Science.

[10] J. Laird. Full abstraction for functional languages with control. In *Proceedings of the Twelfth International Symposium on Logic In Computer Science, LICS '97*, 1997.

[11] J. Laird. *A Semantic Analysis of Control*. PhD thesis, Department of Computer Science, University of Edinburgh, 1998.

[12] G. McCusker. *Games and full abstraction for a functional metalanguage with recursive types*. PhD thesis, Imperial College London, 1996.

[13] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[14] H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science:Logic at St. Petersburg*, LNCS. Springer-Verlag, 1994.

[15] J. Reynolds. Syntactic control of interference. In *Conf. Record 5^{th} ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.

[16] J. Reynolds. The essence of Algol. In *Algorithmic Languages*, pages 345–372. North Holland, 1981.

[17] J. Riecke and H. Thielecke. Typed exceptions and continuations cannot macro-express each other. In J. Wiedermann, P. van Emde Boas and M. Nielsen, editor, *Proceedings of ICALP '99*, volume 1644 of *LNCS*, pages 635 –644. Springer, 1999.

[18] H. Thielecke. On exceptions versus continuations in the presence of state. In *Proceedings of ESOP 2000*, volume 1782 of *LNCS*. Springer, 2000.

A Universal Characterization of the Closed Euclidean Interval

(EXTENDED ABSTRACT)

Martín H. Escardó
School of Computer Science, University of Birmingham
M.Escardo@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~mhe/

Alex K. Simpson*
LFCS, Division of Informatics, University of Edinburgh
Alex.Simpson@dcs.ed.ac.uk
http://www.dcs.ed.ac.uk/home/als/

Abstract *We propose a notion of interval object in a category with finite products, providing a universal property for closed and bounded real line segments. The universal property gives rise to an analogue of primitive recursion for defining computable functions on the interval. We use this to define basic arithmetic operations and to verify equations between them. We test the notion in categories of interest. In the category of sets, any closed and bounded interval of real numbers is an interval object. In the category of topological spaces, the interval objects are closed and bounded intervals with the Euclidean topology. We also prove that an interval object exists in any elementary topos with natural numbers object.*

1 Introduction

In set theory, one can implement the real numbers in many ways. For example, one can use Dedekind sections or equivalence classes of Cauchy sequences of rational numbers. But *what* is it that one is implementing? Assuming classical logic, either implementation produces a complete Archimedian field and, moreover, any two such fields are isomorphic. In fact, for the purposes of classical analysis, one never uses a particular mathematical *implementation* of the reals. One relies instead on the *specification* of the real-number system as a complete Archimedian field and works axiomatically. The only purpose of particular implementations is to be reassured that there is at least one such field.

Unfortunately, when one tries to carry out such a programme in other foundational settings, difficulties arise. One obstacle is that the categoricity of this axiomatization relies on the principle of excluded middle, which is not always available, particularly in settings that are relevant to the theory of computation. Further, one may criticize the axiomatization on the grounds that, although it is aiming to characterize the real line, which is fundamentally a geometric structure, it makes essential use of abstract concepts, such as suprema of bounded sets of points, whose geometric meaning is unclear. In addition, the field axioms involve operations, such as multiplication and reciprocation, which one might rather see as derived from more primitive constructions.

A further objection to the field axiomatization is its lack of explicit computational content. To develop a theory of computability in the sense of Turing [32], one has to start by effectively presenting a particular implementation of the field of real numbers. For example, one can implement real numbers as Cauchy sequences of rational numbers with fixed rate of convergence [3]. Then one has to argue that the basic field operations are computable and that various methods of defining new functions from old preserve computability—see e.g. Weihrauch [34]. With this approach, computability arguments involve heavy manipulation of Gödel numberings, which are detached from the usual practice of real analysis.

The above contrasts with the natural numbers, where primitive recursion, the basic computational mechanism, is not only embodied in their usual Peano axiomatization but can also be taken as their defining property. An elegant formulation of such an axiomatization was given by Lawvere in his definition of a natural numbers object [22]. This style of axiomatization has been adopted for other inductively defined data types, such as lists and trees, which admit canonical forms of recursion that reflect their characterization as initial algebras. Dually, infinite data types, such as streams, are characterized as final coalgebras, with corresponding forms of corecursion. This formulation of data types has been convincingly exploited by Bird and de Moor in their algebraic approach to programming [2].

To place the real numbers into the above framework, one requires a notion of real number data type whose defining property embodies primitive mechanisms for recursion over the reals. In this paper, we present such an axiomatization for closed and bounded line segments, or *interval objects* for short. We characterize interval objects by a universal property that captures a basic geometrical notion and si-

*Research supported by EPSRC grant GR/K06109

0-7695-1281-X/01 $10.00 © 2001 IEEE

multaneously provides a computational notion of recursion. Thus, remarkably, our axiomatization reconciles geometrical and computational conceptions of the line.

In brief, our axiomatization:

(i) is based on elementary geometrical considerations,

(ii) has direct computational content,

(iii) applies in a wide variety of settings,

(iv) gives what one would expect in specific examples.

Regarding (i), we take a *midpoint* operation as the basic structure of line segments, with four axioms that correspond to intuitive geometric properties. We define a *convex body* as a midpoint algebra in which the midpoint operation can be *infinitely iterated*, in a precise sense discussed in the technical development that follows. Then an *interval object* is defined to be a free convex body over two generators, its endpoints. Geometrically, the free property amounts to the fact that any two points of a convex body are connected by a unique line segment.

Regarding (ii), the free property gives rise to an analogue of primitive recursion for defining computable functions on the interval. In particular, we use this to define basic arithmetic operations and to verify equations between them.

Regarding (iii), we make as few ontological commitments as possible by formulating our definitions in the general setting of a category with finite products. Nevertheless, to make the paper accessible to readers who are uncomfortable with category theory, we use, as far as possible, standard algebraic notation, so that everything we say can be easily understood in familiar mathematical terms. Indeed, when specialized to categories such as sets and topological spaces, our definitions assume rather concrete meanings.

Regarding (iv), we have: (1) In the category of sets, any closed and bounded interval of real numbers is an interval object (Theorem 1). (2) In the category of topological spaces, any closed and bounded interval under the usual Euclidean topology is an interval object (Theorem 2). Thus, our axiomatization of line segments exhibits the Euclidean topology as intrinsic rather than imposed structure, because it is this topology that gives rise to an interval object. This is interesting in connection with the often cited fact that the computable functions on the reals are continuous. (3) In any elementary topos with natural numbers object, an interval object is given by the Cauchy completion of the interval of Cauchy reals within the Dedekind reals (Theorem 3). In many cases this coincides with the Cauchy or Dedekind intervals; but, in general, we seem to be identifying an intriguing new intuitionistic notion of real number. For details see Section 9. Some other possible settings are discussed briefly in Section 10.

For lack of space, all proofs are omitted from this extended abstract.

Related work This paper has its origins in the first author's work on exact real number computation [10, 11]. In this approach, real numbers are represented by concrete computational structures such as streams, allowing computations to be performed to any desired degree of accuracy [35, 6, 4, 5, 33]. Of particular relevance to our work is the issue of obtaining an abstract data type of real numbers, in which the underlying computational representation is hidden [5, 8, 10, 11].

In the programming language Real PCF [10], the abstract data type is based on simple real number *constructors* and *destructors*. Mathematically, the constructors are unary midpoint operations $x \mapsto 0 \oplus x$ and $x \mapsto x \oplus 1$ on the unit interval $[0, 1]$, where $x \oplus y = (x + y)/2$ is the binary midpoint operation. These primitives are used by Escardó and Streicher [11] to characterize the interval data type by a universal property, from which structural recursion mechanisms for real numbers are obtained. Thus, this work achieves many of the aims of the present paper. However, it crucially relies on general recursion and the consequent presence of partiality. Indeed, the interval data type includes *partial* real numbers as essential ingredients of its characterization, and the characterization only works in a domain-theoretic setting.

The goal of the present work is to obtain a characterization of the real numbers that applies to a variety of computational settings, including those, such as intuitionistic type theory [25], in which only total functions are available. Although such a programme has not been undertaken previously, algebraic and coalgebraic techniques, similar to the ones used in the present paper, do occur in previous axiomatizations of the reals.

Higgs [14] defines *magnitude algebras* and proves that the interval $[0, \infty]$ endowed with the function $x \mapsto x/2$ and the summation operation $\sum : [0, \infty]^\omega \to [0, \infty]$ is the magnitude algebra freely generated by 1. His definition is purely equational and is based on binary expansions of numbers. Although our work has some connections with Higgs', especially regarding the idea of using an infinitary operation, there are some important differences. Firstly, in the category of topological spaces, the free magnitude algebra over one generator is the interval $[0, \infty]$ with the topology of lower semicontinuity rather than the Euclidean topology. Indeed, the infinitary summation operation is not continuous with respect to the Euclidean topology. Secondly, in general, the Dedekind or Cauchy $[0, \infty]$ intervals in an elementary topos are not magnitude algebras, let alone free ones, as there are toposes, such as Johnstone's topological topos [17], in which these objects do not support the summation operation.

Motivated by the stream implementations of real numbers, Pavlović and Pratt [29] consider *coalgebraic* definitions of the reals. However, they do not make connections

with the computational and geometrical requirements discussed above. Peter Freyd [12] considers a more geometrical coalgebraic approach. In fact, he also places emphasis on midpoint algebras, although the midpoint operation is derived rather than primitive. His approach does appear to have some computational content, but this has yet to be elaborated.

2 Convex bodies and interval objects

This section presents the main definitions of this paper, the notions of *abstract convex body* and *interval object*.

As discussed in the introduction, we define the interval as the free convex body over two generators. To do this, we require an abstract notion of convex body that makes no reference to real numbers. We achieve this by viewing convex bodies as algebraic structures.

The algebraic structure we identify is that associated with the basic ruler-and-compass construction of bisecting a line. Given two points in a convex body A, this construction finds the point midway between them. It thus corresponds to a binary *midpoint* operation $m : A \times A \to A$. We begin by axiomatizing the equational properties of such midpoint operations.

Let $\mathcal{C}$ be a category with finite products.

Definition 2.1 (Midpoint algebra) A *midpoint algebra* in $\mathcal{C}$ is a pair (A, m), where $A \times A \xrightarrow{m} A$ is any morphism, satisfying:

1. $m(x,x) = x$ (*idempotency*)
2. $m(x,y) = m(y,x)$ (*commutativity*)
3. $m(m(x,y), m(z,w)) = m(m(x,z), m(y,w))$ (*transposition*)

A midpoint algebra is said to be *cancellative* if it satisfies:

4. $m(x,z) = m(y,z)$ implies $x = y$ (*cancellation*)

A *homomorphism* from (A,m) to (A',m') is a morphism $A \xrightarrow{f} A'$ such that $f(m(x,y)) = m'(f(x), f(y))$. We write $MidAlg(\mathcal{C})$ for the category of midpoints algebras and their homomorphisms.

In order to understand such ordinary algebraic notation in an arbitrary category with finite products, the variables must be interpreted as generalized elements. Thus, for example, the homomorphism equation states: for all generalized elements $x, y : Z \longrightarrow A$ (where Z is any object), $f \circ m \circ \langle x, y\rangle = m' \circ \langle f \circ x,\ f \circ y\rangle$. In this case, the condition simplifies to the (unquantified) equation $f \circ m = m \circ (f \times f)$.

The equations of midpoint algebras are not new. For example, they have appeared as the axioms of *medial means* in the work of Kermit [20]. They have also recently been popularized by Peter Freyd in his investigations of (co)algebraic properties of the interval [12].

Example 2.2 The set $\mathbb{R}^n$ is a cancellative midpoint algebra under the function $\oplus : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ defined by

$$\mathbf{x} \oplus \mathbf{y} \quad = \quad (\mathbf{x} + \mathbf{y})/2.$$

This yields a whole range of cancellative midpoint algebras given by subsets $A \subseteq \mathbb{R}^n$ closed under $\oplus$. We call such midpoint algebras *standard midpoint subalgebras of* $\mathbb{R}^n$. Examples are: the set of dyadic rational points; the set of rational points; the set of algebraic points; any convex set.

These examples show that the midpoint axioms are still far from capturing the full power of convexity, which requires one to be able to fill in an entire connected line between any two points. Intuitively, we need to express something like a notion of Cauchy completeness for midpoint algebras. However, Cauchy completeness itself cannot be the appropriate notion, as midpoint algebras do not necessarily carry a metric structure. More fundamentally, we cannot use the notion of metric space to define the interval, because axiomatizing metric spaces already begs the question of what the real numbers are. Instead, we need a method of axiomatizing the completeness of midpoint algebras in terms of their algebraic structure alone.

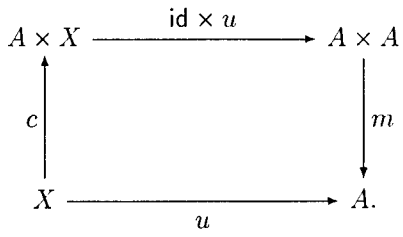
Consider an arbitrary sequence of points $x_0, x_1, \ldots$ in an ordinary Euclidean convex body A. Let z be any point of A and consider the derived sequence

$$m(x_0, z),\ m(x_0, m(x_1, z)),\ m(x_0, m(x_1, m(x_2, z))),\ \ldots$$

If A is bounded then this is a Cauchy sequence whose unique limit point lies in A and is independent of z. Thus, any sequence $x_0, x_1, \ldots$, determines a unique point $m(x_0, m(x_1, m(x_2, \ldots)))$ obtained by infinitely iterating the binary operation m over the sequence. Our notion of completeness for a midpoint algebra A is to ask that such infinite iterations always exist.

In the category of sets, such a requirement can be expressed directly, albeit clumsily—see Proposition 3.1. Remarkably, there is a very concise formulation in purely categorical terms. Infinite sequences of elements of A are naturally expressed using coalgebras for the functor $(A \times (-))$, i.e. morphisms of the form $\langle h, t\rangle : X \longrightarrow A \times X$. Indeed, any such coalgebra determines an object X of sequences of elements of A, as specified by the *head* and *tail* maps $h : X \longrightarrow A$ and $t : X \longrightarrow X$ respectively. We can now state the property of being able to iterate the midpoint operation m over any sequence so specified.

Definition 2.3 (Iterative algebra) A midpoint algebra (A, m) is *iterative* if it satisfies the *iteration axiom*: for every map $X \xrightarrow{c} A \times X$, there exists a unique $X \xrightarrow{u} A$ such that the diagram below commutes.



In other words, (A, m) is iterative if, for any coalgebra $c = \langle h, t \rangle : X \longrightarrow A \times X$, there exists a unique u satisfying $u(x) = m(h(x), u(t(x)))$.

The above definition states that a midpoint algebra (A, m) is iterative if it is final as an $(A \times (-))$-algebra with respect to coalgebra-to-algebra homomorphisms from $(A \times (-))$-coalgebras. Interestingly, the dual notion of a coalgebra being initial with respect to arbitrary algebras has arisen in recent work of Taylor [31, Section 6.3] and Eppendahl [9].

We are now in a position to formulate our abstract notion of convex body.

Definition 2.4 (Abstract convex body) An *abstract convex body* is a cancellative iterative midpoint algebra.

We henceforth omit the word abstract, except when required to avoid confusion due to alternative notions of convex body being available (for example, in Euclidean space, where ordinary convex bodies are convex sets with nonempty interior). We write $Conv(\mathcal{C})$ for the full subcategory of $MidAlg(\mathcal{C})$ whose objects are convex bodies.

Example 2.5 Continuing from Example 2.2, any bounded convex subset of $\mathbb{R}^n$, considered as a standard midpoint subalgebra of $\mathbb{R}^n$, is an abstract convex body. Indeed, given functions $h : X \to A$ and $t : X \to X$, where X is any set, the unique function $u : X \to A$ determined from the coalgebra $\langle h, t \rangle : A \to A \times X$ by the iteration axiom is

$$u(x) \quad = \quad \sum_{i \geq 0} 2^{-(i+1)} h(t^i(x)). \qquad (1)$$

An important point is that the boundedness of A is crucial for u to be well-defined. In fact, a standard midpoint subalgebra of $\mathbb{R}^n$ is an abstract convex body if and only if it is a bounded convex subset of $\mathbb{R}^n$; and, given a bounded convex subset B of $\mathbb{R}^m$, a function $f : A \to B$ is a homomorphism of abstract convex bodies (i.e. a homomorphism w.r.t. $\oplus$) if and only if it is affine. See Section 3 for details.

Example 2.6 Let A be any bounded convex subset of $\mathbb{R}^n$ endowed with the Euclidean topology. Then $\oplus$ also exhibits A as a convex body in the category **Top** of topological spaces. Indeed, given any continuous $(A \times (-))$-coalgebra $\langle h, t \rangle : X \to A \times X$ (where X is any space), the function u defined in (1) is again the unique map required by the iteration axiom. The interesting fact here is that u is continuous. This example will be expanded upon in Section 8.

As motivated in the introduction, the interval will be defined as the free abstract convex body over two generators. This amounts to being an initial object in a suitable category of bipointed convex bodies.

A *bipointed* convex body is a structure (A, m, a, b) where (A, m) is a convex body and $a, b : \mathbf{1} \longrightarrow A$ are global points. *Homomorphisms* between bipointed convex bodies are required to preserve the points as well as the binary algebra structure; i.e. $f : A \longrightarrow A'$ is a homomorphism from (A, m, a, b) to (A', m', a', b') if and only if it is a homomorphism from (A, m) to (A', m') and $a' = f \circ a$ and $b' = f \circ b$. We write $BiConv(\mathcal{C})$ for the category of bipointed convex bodies and their homomorphisms.

We can now give the main definition of the paper.

Definition 2.7 (Interval object) An *interval object* in $\mathcal{C}$ is an initial object in $BiConv(\mathcal{C})$.

Example 2.8 In **Set**, any closed interval $[a, b] \subseteq \mathbb{R}$, with $a < b$, gives an interval object $([a, b], \oplus, a, b)$. Of course the choice of a and b makes no difference. For future convenience, we take the interval $\mathbb{I} = [-1, 1]$ as our standard closed interval and $(\mathbb{I}, \oplus, -1, 1)$ as our standard interval object. This example is discussed in more detail in Section 3.

Example 2.9 In **Top**, $(\mathbb{I}, \oplus, -1, 1)$ is again an interval object when $\mathbb{I}$ is equipped with the Euclidean topology. This is discussed further in Section 8.

3 Interval objects in the category of sets

In this section we study abstract convex bodies in the category **Set** of sets, and we show that the interval object in **Set** is indeed $(\mathbb{I}, \oplus, -1, 1)$, as claimed in Example 2.8.

The least familiar aspect of the definition of convex body is the notion of iterative algebra. We begin by showing that, in **Set**, iterative algebras are exactly algebras supporting an additional operation of countably-infinite arity that satisfies certain characterising properties relating it to the binary operation. In general, this reformulation provides the most straightforward method of showing that an algebra is iterative.

Proposition 3.1 *Let (A, m) be a midpoint algebra in* **Set**.

1. *(A, m) is iterative if and only if there exists a function $M : A^\omega \to A$ satisfying:*

   (a) $M(x_0, x_1, x_2, \ldots) = m(x_0, M(x_1, x_2, x_3, \ldots))$

   (b) If $y_0 = m(x_0, y_1)$, $y_1 = m(x_1, y_2)$, $y_2 = m(x_2, y_3)$, *... then* $y_0 = M(x_0, x_1, x_2, \ldots)$.

   Moreover if (A, m) is iterative then there is a unique M satisfying (a).

2. *If (A, m) and (A', m') are iterative midpoint algebras then any homomorphism $f : A \to A'$ is also a homomorphism with respect to the associated infinitary M and M'; i.e. for every sequence $x_0, x_1, \ldots$,*

$$f(M(x_0, x_1, \ldots)) = M'(f(x_0), f(x_1), \ldots).$$

With an appropriate reformulation, the above proposition generalizes from the category of sets to any category with finite products and a parameterized natural numbers objects.

It is useful to identify additional equational properties satisfied by the the associated infinitary operations. We use $M_i(x_i)$ as a shorthand for $M(x_0, x_1, x_2, \ldots)$.

Proposition 3.2 *For any iterative midpoint algebra (A, m) in* **Set**, *with infinitary $M : A^\omega \to A$,*

1. $x = M(x, x, x, \ldots)$,
2. $m(x, y) = M(x, y, y, y, \ldots)$,
3. $M_i(M_j(x_{ij})) = M_j(M_i(x_{ji}))$,
4. $M_i(m(x_i, y_i)) = m(M_i(x_i), M_i(y_i))$.

For an iterative midpoint algebra to be a convex body it must also be cancellative. We have yet to see any technical consequence of this property. In fact, for iterative midpoint algebras, cancellation is equivalent to an important approximation property. To formulate this, we write m_n for the $(n+1)$-ary operation defined by $m_0(x) = x$ and $m_n(x_0, \ldots, x_n) = m(x_0, m_{n-1}(x_1, \ldots, x_n))$ for $n \geq 1$. Thus m_1 is just m itself.

Proposition 3.3 *For an iterative midpoint algebra (A, m) in* **Set**, *the following are equivalent.*

1. *(A, m) is cancellative.*
2. *The associated $M : A^\omega \to A$ satisfies the following approximation property.*

   If, for all $n \geq 0$, there exist $z_n, w_n \in A$ such that $m_n(x_0, \ldots, x_{n-1}, z_n) = m_n(y_0, \ldots, y_{n-1}, w_n)$ then

$$M(x_0, x_1, \ldots) = M(y_0, y_1, \ldots).$$

This is far from immediate and is used crucially in the proof of Theorem 1.

Having obtained a good understanding of what the different aspects of the definition of convex body mean in **Set**, we return to Examples 2.5 and 2.8.

Proposition 3.4 *If A is a standard midpoint subalgebra of $\mathbb{R}^n$, then A is an abstract convex body if and only if it is a bounded convex subset of $\mathbb{R}^n$.*

Suppose $A \subseteq \mathbb{R}^n$ and $A' \subseteq \mathbb{R}^m$ are convex sets. Recall that a function $f : A \to A'$ is said to be *affine* if it preserves so-called convex combinations, i.e., for $\lambda_1, \ldots, \lambda_k \in [0, 1]$ with $\sum_{i=1}^k \lambda_i = 1$,

$$f(\sum_{i=1}^k \lambda_i \mathbf{x}_i) \quad = \quad \sum_{i=1}^k \lambda_i f(\mathbf{x}_i).$$

The next proposition demonstrates the naturalness of homomorphisms between abstract convex bodies.

Proposition 3.5 *For bounded convex sets $A \subseteq \mathbb{R}^n$ and $A' \subseteq \mathbb{R}^m$, a function $f : A \to A'$ is affine if and only if it is a homomorphism with respect to $\oplus$.*

An example due to Peter Freyd [12], which uses the axiom of choice, can be used to show that the boundedness assumption is essential for Proposition 3.5 to hold.

Theorem 1 $(\mathbb{I}, \oplus, -1, 1)$ *is an interval object in* **Set**.

4 Parameterized interval objects

It is well known that Lawvere's elegant definition of a natural numbers object, which works very well in cartesian closed categories, is not powerful enough in categories with weaker structure. Instead, a modified parameterized definition is needed [21, 7]. In a category with finite products, the notion of parameterized natural numbers object supports the definition of functions by primitive recursion. Moreover, in a cartesian closed category, any ordinary natural numbers objects is automatically parameterized. Much the same situation arises for interval objects.

Definition 4.1 (Parameterized interval object) A *parameterized interval object* is a bipointed convex body $(I, \oplus, -1, 1)$ such that, for any convex body (A, m) and morphisms $X \xrightarrow{f} A$ and $X \xrightarrow{g} A$ in $\mathcal{C}$, there exists a unique morphism $X \times I \xrightarrow{(\!|f,g|\!)} A$ satisfying

$$\begin{aligned} (\!|f,g|\!)(x, y \oplus z) &= m((\!|f,g|\!)(x,y), (\!|f,g|\!)(x,z)), \\ (\!|f,g|\!)(x,-1) &= f(x), \\ (\!|f,g|\!)(x,1) &= g(x), \end{aligned}$$

i.e. there is a unique *right-homomorphism* of bipointed convex bodies from $X \times I$ to A.

By instantiating X to the terminal object, it is easily seen that any parameterized interval object is indeed an interval object. The converse holds when $\mathcal{C}$ is cartesian closed:

Proposition 4.2 *If $\mathcal{C}$ is cartesian closed then any interval object is parameterized.*

Henceforth in this section, let $\mathcal{C}$ be a category with finite products and parameterized interval object $(I, \oplus, -1, 1)$. The basic arithmetic operations on I can be defined by

$$\begin{aligned} \mathbf{1} \xrightarrow{0} I &= (-1) \oplus (1), \\ I \xrightarrow{-} I &= (\!|1, -1|\!), \\ I \times I \xrightarrow{\times} I &= (\!|-, \mathsf{id}_I|\!). \end{aligned}$$

More explicitly, the above defines multiplication as the unique morphism $I \times I \xrightarrow{\times} I$ satisfying

$$\begin{aligned} x \times (y \oplus z) &= (x \times y) \oplus (x \times z), \\ x \times (-1) &= -x, \\ x \times 1 &= x. \end{aligned}$$

Importantly, the universal property of I, stated in Definition 4.1, suffices to establish the basic equations between the above operations.

Proposition 4.3 $--x = x$,
$x \times y = y \times x$,
$x \times (y \times z) = (x \times y) \times z$,
$-0 = 0$,
$x \oplus -x = 0$,
$-(x \oplus y) = (-x) \oplus (-y)$,
$x \times 0 = 0$,
$x \times -y = -(x \times y)$.

The most entertaining proof is that of the commutativity of multiplication.

5 Primitive interval functions

In this section we give some preliminary results on the power of the notion of interval object with respect to defining functions on the interval. As mentioned above, any parameterized natural numbers object supports definition by primitive recursion. Here we investigate the definitional mechanisms supported by parameterized interval objects.

In fact, a parameterized interval object supports two complementary styles of definition. On the one hand, the universal property of parameterized initiality gives one mechanism for defining functions, used above to define negation and multiplication. On the other, the couniversal property of the iteration axiom supports another type of definition, needed, for example, to define non dyadic rational numbers. Parameterized interval objects allow any combination of these two styles. We investigate the power of such combinations for the purpose of defining functions on $\mathbb{I}$ in **Set**.

Definition 5.1 (Primitive interval functions) The *primitive interval functions* on $\mathbb{I}$ are the functions in the smallest family $\{\mathcal{F}_n \subseteq \mathbb{I}^n \to \mathbb{I}\}_{n \geq 0}$ satisfying:

(i) $-1, 1 \in \mathcal{F}_0$.

(ii) If $f \in \mathcal{F}_m$ and $g_1, \ldots, g_m \in \mathcal{F}_n$ then the composite $f \circ \langle g_1, \ldots, g_m \rangle \in \mathcal{F}_n$.

(iii) If $f, g \in \mathcal{F}_n$ then the function h defined below is in $\mathcal{F}_{n+1}$:

$$h(\mathbf{x}, y) = \frac{1}{2}(1 - y)f(\mathbf{x}) + \frac{1}{2}(1 + y)g(\mathbf{x}).$$

(iv) If $f_1, \ldots, f_n, g \in \mathcal{F}_n$ then the unique function h satisfying the equation below is in $\mathcal{F}_n$:

$$h(\mathbf{x}) = \frac{1}{2}g(\mathbf{x}) + \frac{1}{2}h(f_1(\mathbf{x}), \ldots, f_n(\mathbf{x})).$$

Here (iii) corresponds to the parameterized initiality of $\mathbb{I}$, with respect to $\mathbb{I}^n$ as the object of parameters, and (iv) corresponds to the iteration axiom, as induced by the coalgebra $\langle g, f_1, \ldots, f_n \rangle : \mathbb{I}^n \to \mathbb{I} \times \mathbb{I}^n$. Note that property (ii) means that tuples of primitive interval functions between finite powers of $\mathbb{I}$ form a category. This category has finite products because the projections are definable, using (iii).

The function defined by (iv) is given explicitly by

$$h(\mathbf{x}) = \sum_{i \geq 0} 2^{-(i+1)} g(\langle f_1, \ldots, f_n \rangle^i(\mathbf{x})).$$

A natural generalization is to replace the sequence $(g \circ \langle f_1, \ldots, f_n \rangle^i)_i$ of composite functions with an arbitrary sequence of (already defined) n-ary functions.

Definition 5.2 (Countably-primitive functions) The *countably-primitive interval functions* on $\mathbb{I}$ are the functions in the smallest family $\{\mathcal{F}_n \subseteq \mathbb{I}^n \to \mathbb{I}\}_{n \geq 0}$ satisfying (i)–(iii) of Definition 5.1 and also

(iv)′ Given $f_0, f_1, \ldots \in \mathcal{F}_n$, the function h defined below is in $\mathcal{F}_n$:

$$h(\mathbf{x}) = \sum_{i \geq 0} 2^{-(i+1)} f_i(\mathbf{x}).$$

Clearly every primitive interval function is a countably-primitive interval function. The converse does not hold as there are continuum many countably-primitive interval functions, but only countably many primitive interval functions. Indeed, every element of $\mathbb{I}$ gives a countably-primitive interval function of arity 0 (i.e. a constant). Although this cannot hold for the primitive interval functions, we do at least have the following.

Proposition 5.3 *Every rational in $\mathbb{I}$ gives a primitive interval constant.*

The proof makes crucial use of property (iv).

As in Section 4, we have $\oplus, -, \times$ as primitive interval functions. Thus every n-variable $\oplus$-polynomial (i.e. polynomial where $\oplus$ replaces the usual $+$) with rational coefficients is an n-ary primitive interval function.

We are not sure how much further definability can be pushed with the primitive interval functions, as we now show that even the countably-primitive interval functions are very limited.

Proposition 5.4 *If f is an n-ary countably-primitive interval function, and $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1} \in \mathbb{I}$ are such that $y_i = x_i$ whenever $x_i \in \{-1, 1\}$, then $f(x_0, \ldots, x_{n-1}) \in \{-1, 1\}$ implies $f(y_0, \ldots, y_{n-1}) = f(x_0, \ldots, x_{n-1})$.*

This is proved by induction over the defining properties of the countably-primitive interval functions.

Thus if f is a unary countably-primitive interval function and $f(x) \in \{-1, 1\}$ for some x in the interior $(-1, 1)$ then f is a constant function. Clearly then, the following *truncated double* function is not a countably-primitive interval function.

$$d(x) = \begin{cases} 1 & \text{if } 1/2 \leq x, \\ 2x & \text{if } -1/2 \leq x \leq 1/2, \\ -1 & \text{if } x \leq -1/2. \end{cases}$$

Accordingly, define the *d-primitive interval functions* to be the smallest class of functions containing d and closed under (i)–(iv). Define the *countably-d-primitive interval functions* analogously. The reason for selecting d amongst the non-countably-primitive interval functions is:

Proposition 5.5 *The n-ary countably-d-primitive interval functions are exactly the continuous functions $\mathbb{I}^n \to \mathbb{I}$.*

The proof uses the Stone-Weierstrass approximation theorem [30].

Thus including d as a basic function enormously increases definability. It is our hope that this increase in definability also means that the d-primitive interval functions form a useful class, somewhat analogous to the primitive recursive functions on $\mathbb{N}$. Although we have yet to undertake any systematic investigation of this class, we do have one important result. Recall the standard notion of an n-ary *computable function* on $\mathbb{I}$ [34].

Proposition 5.6 *Every n-ary d-primitive interval function is an n-ary computable function on $\mathbb{I}$.*

This result follows from Theorem 3 of Section 9 below, by interpreting it in a realizability topos in which the morphisms on the interval are exactly the computable functions. However, in the next section, we outline a direct proof, by showing that the computable functions are closed under the defining properties of the d-primitive interval functions.

6 An interval data type

In Proposition 3.1, we have seen that, in the category of sets, the iteration axiom is captured by the existence of an infinitary version M of the midpoint operation m. Moreover, a function of convex bodies is a homomorphism with respect to m if and only if it is a homomorphism with respect to M. Additionally, Proposition 3.2 shows that m is easily defined from M. This suggests that one might consider the ω-ary operation M as the primitive algebraic operator on convex bodies, rather than m. In this section, we exploit this idea to base a data type for the interval $\mathbb{I}$ on the term algebra of an ω-ary operation M and two constants -1 and 1.

We outline an implementation using a functional programming notation similar to ML [28] and Haskell [1] (it is not important whether an eager or lazy language is used). Our data type `I` is defined as follows.

```
datatype I = -1 | 1 |  M of Nat -> I
```

Within the interval type `I`, we single out the ω-branching well-founded trees as those data elements representing points of the interval. Such trees are precisely the elements of the term algebra mentioned above. To interpret a tree as representing an element of $\mathbb{I}$, the infinitary operator M is interpreted as the iterated midpoint operation

$$M(x_0, x_1, x_2, \ldots) = \sum_{i=0}^{\infty} 2^{-(i+1)} x_i,$$

using which any ω-branching well-founded tree evaluates to a unique point in $\mathbb{I}$. Thus, by this interpretation, $\mathbb{I}$ is given as a quotient of the set of all ω-branching well-founded trees.

The iteration axiom of Definition 2.3, in the concrete form given in Example 2.5, corresponds to the following *corecursion combinator*.

```
corec : (X -> I) -> (X -> X) -> (X -> I)

corec h t x = M (\i -> h(t^i(x)))
```

In this definition, `\i->t` is typewriter notation for the lambda expression $\lambda i.t$ and we use the evident notation for function iteration.

The initiality of $\mathbb{I}$, as in Definition 2.7, is exhibited by the following *recursion combinator*.

```
rec : ((Nat -> A) -> A) -> A -> A -> (I -> A)

rec N a b  -1 = a
rec N a b   1 = b
rec N a b  (M s) = N (\i -> rec N a b (s i))
```

In this definition, the first argument N is the infinitary midpoint operation of a given bipointed convex body A, and the second and third arguments a and b are the distinguished points. We have not built any explicit type of parameters into the type of `rec`, because parameterization is induced automatically by the functional language. For example, negation and multiplication are defined as in Section 4, using the recursion combinator.

```
neg : I -> I
neg = rec M  1 -1

mul : I -> I -> I
mul x = rec M (neg x) x
```

The recursion and corecursion combinators correspond to conditions (iii) and (iv) of Definition 5.1 respectively. The truncated double function can also be implemented using the datatype `I`, but this is surprisingly tricky. However, curiously, an algorithm for doing this occurs fairly explicitly in our (omitted) proof of Theorem 3 below. It follows that the d-primitive interval functions are definable on our interval data type `I`.

Because we are using a non-standard representation of the interval, based on the infinitary midpoint operation, it is important to show that our representation is interconvertible with the standard representations used in exact real number arithmetic. One such representation, *signed binary*, uses a data type `I'` of infinite sequences of the three digits -1, 0 and 1—see [35]. It is trivial to convert from signed binary sequences to our representation `I`, using the facts that $0 = M(-1,1,1,1,\ldots)$ and that a signed binary expansion $0.d_0d_1d_2\ldots$ is the same as $M(d_0,d_1,d_2,\ldots)$. To translate in the other direction, one first defines the iterated midpoint operation `M' : (Nat->I') ->I'` (an interesting programming exercise), and then the conversion function `I->I'` is simply `rec M' (\i -> -1) (\i -> 1)`.

Although we have written this section using a functional language with general recursion, we remark that our representation of the interval can be implemented even more directly using intuitionistic type theory [25]. Indeed, by formulating the recursive definition of the data type `I` as a W-type, one obtains precisely the well-founded ω-branching trees over -1 and 1, and our recursion combinator is simply the recursor for this type.

7 Basic categorical properties

In this section, we turn our attention to general properties of convex bodies and interval objects arising from their categorical definitions. This general investigation will be useful in Sections 8 and 9, in which we study examples in categories other than **Set**.

One benefit of having simple abstract definitions of convex body and interval object is that it is easy to prove that these notions are preserved by various categorical constructions and functors. In this section, we state basic results of this nature. The proofs are all routine.

As in Section 2, let $\mathcal{C}$ be a category with finite products.

Proposition 7.1 *The forgetful functors $Conv(\mathcal{C}) \to \mathcal{C}$ and $BiConv(\mathcal{C}) \to \mathcal{C}$ create limits.*

In particular, if (A,m) and (A',m') are convex bodies then so is $A \times A'$ endowed with

$$(A\times A')\times(A\times A') \xrightarrow{\cong} (A\times A)\times(A'\times A') \xrightarrow{m\times m'} A\times A'$$

and an analogous statement holds for bipointed convex bodies. One simple consequence of this result is that, for any interval object $(I,\oplus,a,b)$, the n-dimensional cube I^n has an induced convex body structure.

As well as being closed under limits, convex bodies are also closed under internal powers.

Proposition 7.2 *If (A,m) is a convex body then so is*

$$(A^B,\ A^B \times A^B \xrightarrow{\cong} (A\times A)^B \xrightarrow{m^B} A^B)$$

for any exponentiable object B.

Again, the analogous result holds for bipointed convex bodies.

It is also straightforward to establish conditions under which (bipointed) convex bodies are preserved by functors. Suppose $\mathcal{D}$ is a category with finite products, and the functor $F : \mathcal{C} \to \mathcal{D}$ preserves finite products. Then there is a functor $\overline{F} : MidAlg(\mathcal{C}) \to MidAlg(\mathcal{D})$ whose action on objects is:

$$\overline{F}(A,m) = (FA,\ FA\times FA \xrightarrow{\cong} F(A\times A) \xrightarrow{Fm} FA)$$

and whose action on morphisms is inherited from F.

Proposition 7.3 *Suppose that F has a left adjoint.*

1. *The functor $\overline{F} : MidAlg(\mathcal{C}) \to MidAlg(\mathcal{D})$ cuts down to a functor $\overline{F} : Conv(\mathcal{C}) \to Conv(\mathcal{D})$. Similarly, by extending the action of $\overline{F}$ to bipointed objects, a functor $\overline{F} : BiConv(\mathcal{C}) \to BiConv(\mathcal{D})$ is obtained.*
2. *If $F : \mathcal{C} \to \mathcal{D}$ also has a right adjoint $G : \mathcal{D} \to \mathcal{C}$ then $\overline{G} : Conv(\mathcal{D}) \to Conv(\mathcal{C})$ is right adjoint to the functor $\overline{F} : Conv(\mathcal{C}) \to Conv(D)$, and $\overline{G} : BiConv(\mathcal{D}) \to BiConv(\mathcal{C})$ is right adjoint to $\overline{F} : BiConv(\mathcal{C}) \to BiConv(D)$. Thus, in particular, $F : \mathcal{C} \to \mathcal{D}$ preserves interval objects.*

It follows from 1 above that if $\mathcal{C}$ is a full reflective subcategory of $\mathcal{D}$ and if $\mathcal{D}$ has an interval object $(I,\oplus,-1,1)$ where I is an object of $\mathcal{C}$ then $(I,\oplus,-1,1)$ is also an interval object in $\mathcal{C}$.

A special case of statement 2 is that interval objects are preserved by the inverse image functors of essential geometric morphisms between elementary toposes. Thus if $f : \mathcal{E} \to \mathcal{E}'$ is an essential geometric morphism and $\mathcal{E}'$ has an interval object then its image under f^* gives an interval object in $\mathcal{E}$. In particular, by Theorem 1, every presheaf topos $\mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$ has an interval object obtained as $\Delta(\mathbb{I})$ — recall that the constant presheaf functor, $\Delta : \mathbf{Set} \to \mathbf{Set}^{\mathcal{C}^{\mathrm{op}}}$, is the inverse image functor of an essential geometric morphism [24]. More generally, in Section 9, we show that any elementary topos with natural numbers object has an interval object.

8 Interval objects in the category of topological spaces

In this section we return to the claims made earlier in Examples 2.6 and 2.9, investigating abstract convex bodies and interval objects in the category **Top** of topological spaces.

Proposition 3.1 generalizes to **Top** with the requirement that $M : A^\omega \to A$ be continuous with respect to the product topology. It follows that, for a bounded convex $A \subseteq \mathbb{R}^n$, the midpoint algebra $(A, \oplus)$ with the discrete topology is *not* an abstract convex body in **Top** , because this topology does not make the iterated midpoint operation into a continuous function. Thus the notion of abstract convex body forces one to consider more reasonable topologies on $(A, \oplus)$.

Proposition 8.1 *For any bounded convex subset $A \subseteq \mathbb{R}^n$ endowed with the Euclidean topology, $(A, \oplus)$ is an abstract convex body in* **Top**.

This result is derived from Proposition 3.4, by proving that the infinitary midpoint operation is continuous. Certain other basic information about convex bodies in **Top** can be inferred using Proposition 7.3. The forgetful functor U : **Top** $\to$ **Set** has both a left adjoint Δ (giving the discrete topology) and a right adjoint ∇ (giving the indiscrete topology). Thus, both U and ∇ preserve convex bodies. As U does, we see that, by Proposition 3.4, under any topology whatsoever, for a standard midpoint subalgebra A of $\mathbb{R}^n$ to be a convex body in **Top**, A must be a bounded convex set. Also, for any bounded convex set, $(A, \oplus)$ with the indiscrete topology is a convex body in **Top**.

Also, by Proposition 3.4, if an interval object exists in **Top** then U preserves it. In fact, we have already claimed in Example 2.9 that $(\mathbb{I}, \oplus, -1, 1)$ is an interval object in **Top** when given the Euclidean topology. As **Top** is not cartesian closed, it is appropriate to show that this is a parameterized interval object in the sense of Section 4.

Theorem 2 $(\mathbb{I}, \oplus, -1, 1)$ *with the Euclidean topology is a parameterized interval object in* **Top**.

By Proposition 7.3.1, $(\mathbb{I}, \oplus, -1, 1)$ with the Euclidean topology is a parameterized interval object in any full reflective subcategory of **Top** that contains the closed Euclidean interval. Thus, for example, it is a parameterized interval object in the category of compact Hausdorff spaces.

9 Interval objects in an elementary topos

In this section we prove that an interval object exists in any elementary topos with natural numbers object. There are at least two reasons to be interested in such a result. Firstly, elementary toposes include all Grothendieck and realizability toposes, of which there are numerous examples with direct geometrical and/or computational significance. Indeed, we have already mentioned that the results of this section can be used to prove Proposition 5.6.

Our second motivation is to study the notion of interval object using an intuitionistic background logic. It is well known that intuitionistic logic draws sharp distinctions between different, though classically equivalent, definitions of real number. To better understand our notion of interval object, we compare it to the competing intuitionistic accounts of the interval. Somewhat surprisingly, rather than obtaining one of the established notions, interval objects give rise to an apparently new intuitionistic notion of real number, albeit one that coincides with extant notions under the mild assumption of number-number choice.

Let $\mathcal{E}$ be an elementary topos with natural numbers object $\mathbf{N}$. Among the alternative notions of real number available, two are considered as being the most natural, the *Dedekind* reals $\mathbf{R}_D$ and the *Cauchy* (or *Cantor*) reals $\mathbf{R}_C$. Both are defined using the object of rationals $\mathbf{Q}$ and its associated ordering. The reader is referred to [16] for details.

A basic fact is that one has inclusions

$$\mathbf{Q} \subseteq \mathbf{R}_C \subseteq \mathbf{R}_D.$$

We say that a subobject $X \subseteq \mathbf{R}_D$ is *Cauchy complete* if every Cauchy sequence in $X^{\mathbf{N}}$ (with modulus) has a limit in X. It is easy to see that the Dedekind reals are Cauchy complete. Obviously, the rationals are not Cauchy complete. The Cauchy reals partially rectify the non-completeness of $\mathbf{Q}$ by adding all limits of Cauchy sequences of rationals. Given $\mathbf{N}$-$\mathbf{N}$-choice, this suffices to make $\mathbf{R}_C$ itself Cauchy complete. However, it seems that, in general, $\mathbf{R}_C$ is *not* Cauchy complete, as, given a Cauchy sequence of Cauchy reals, there is no mechanism for selecting representative rational sequences from which the required limiting sequence of rationals can be extracted.

The possible failure of Cauchy completeness for $\mathbf{R}_C$ makes it natural to introduce another object of reals, namely, the *Cauchy completion of* $\mathbf{Q}$ *within* $\mathbf{R}_D$. This object, which we call the object of *Euclidean* reals $\mathbf{R}_E$, is defined as the

intersection of all Cauchy complete subobjects of $\mathbf{R}_D$ containing the rational numbers.

We have identified three objects of reals

$$\mathbf{R}_C \subseteq \mathbf{R}_E \subseteq \mathbf{R}_D.$$

In the case that $\mathcal{E}$ satisfies $\mathbf{N}$-$\mathbf{N}$-choice, both inclusions are equalities. The Grothendieck topos of sheaves over the Euclidean line is a simple example in which the second inclusion is strict. To our embarrassment, we do not know an example in which the first inclusion is strict. Thus we do not know if the envisaged failure of the Cauchy completeness of $\mathbf{R}_C$ is actually possible—although we are sure that it must be.

Each notion of real number object determines a corresponding notion of interval object; for example,

$$\begin{array}{rcl} \mathbf{I}_D & = & \{x \in \mathbf{R}_D \mid -1 \leq x \leq 1\} \\ \mathbf{I}_E & = & \{x \in \mathbf{R}_E \mid -1 \leq x \leq 1\} \quad = \quad \mathbf{R}_E \cap \mathbf{I}_D. \end{array}$$

The reason for introducing the Euclidean reals in the first place is the following.

Theorem 3 $(\mathbf{I}_E, \oplus, -1, 1)$ *is an interval object in* $\mathcal{E}$.

Our proof is very long and makes crucial use of Pataraia's intuitionistic fixed-point theorem for monotonic endomaps of directed complete partial orders [27].

10 Concluding remarks

We have provided an axiomatization of the interval, by means of a geometrically motivated universal property that supports the definition of computable functions. Moreover, we have investigated this axiomatization in a number of settings.

Many other settings remain to be investigated. In the category of setoids over intuitionistic type theory [15, 26], it can be shown that any of the usual constructions of a closed real interval gives an interval object. In the category of locales over any topos, we conjecture that the standard localic interval [18] is an interval object.

By definition, an interval object is a free convex body over two generators. Freely generated convex bodies over different generating objects coincide with other familiar mathematical structures. Interesting examples occur in the category of topological spaces: (1) The free convex body over Sierpinski space is the interval with the topology of lower semicontinuity. (2) The free convex body over the flat domain of booleans under the Scott topology is the interval domain studied in [11] with its pointwise midpoint structure. (3) The free convex body over a finite discrete space of cardinality n is an n-simplex. In particular, the free convex body over three and four generators are the triangle and the tetrahedron. All the above examples are applications of the left adjoint to the forgetful functor from topological convex bodies to topological spaces, which exists by Freyd's Adjoint Functor Theorem [23].

There are intriguing connections between midpoint algebras and the probabilistic algebras that arise in the study of probabilistic powerdomains—see the axiomatizations discussed by Heckmann [13]. It is plausible that the free convex body over a sufficiently nice domain may be nothing but the probabilistic powerdomain of normalized valuations [19].

References

[1] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 2nd edition, 1998.

[2] R. Bird and O. de Moor. *Algebra of programming*. Prentice Hall Europe, London, 1997.

[3] E. Bishop and D. Bridges. *Constructive Analysis*. Springer-Verlag, Berlin, 1985.

[4] H.J. Boehm. Constructive real interpretation of numerical programs. *SIGPLAN Notices*, 22(7):214–221, 1987.

[5] H.J. Boehm and R. Cartwright. Exact real arithmetic: Formulating real numbers as functions. In Turner. D., editor, *Research Topics in Functional Programming*, pages 43–64. Addison-Wesley, 1990.

[6] H.J. Boehm, R. Cartwright, M. Riggle, and M.J. O'Donnel. Exact real arithmetic: A case study in higher order programming. In *ACM Symposium on Lisp and Functional Programming*, 1986.

[7] R.L. Crole. *Categories for Types*. Cambridge University Press, Cambridge, 1993.

[8] P. Di Gianantonio. *A functional approach to computability on real numbers*. PhD thesis, University of Pisa, 1993. Technical Report TD 6/93.

[9] A. Eppendahl. Coalgebra-to-algebra morphisms. *Electronic Notes in Theoretical Computer Science*, 29, 1999.

[10] M.H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, 1996.

[11] M.H. Escardó and Th. Streicher. Induction and recursion on the partial real line with applications to Real PCF. *Theoret. Comput. Sci.*, 210(1):121–157, 1999.

[12] P. Freyd. Public communications to the categories mailing list. `http://www.mta.ca/~cat-dist/categories.html`, 1999–2000.

[13] R. Heckmann. Probabilistic domains. pages 142–156. Springer, LNCS 787, 1994.

[14] D. Higgs. A universal characterization of $[0, \infty]$. *Nederl. Akad. Wetensch. Indag. Math.*, 40(4):448–455, 1978.

[15] M. Hofmann. *Extensional constructs in intensional type theory*. Springer-Verlag London Ltd., London, 1997.

[16] P.T. Johnstone. *Topos Theory*. Academic Press, London, 1977.

[17] P.T. Johnstone. On a topological topos. *Proceedings of the London Mathematical Society*, 38:237–271, 1979.

[18] P.T. Johnstone. *Stone Spaces*. Cambridge University Press, Cambridge, 1982.

[19] C. Jones. *Probabilistic Non-determinism*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, January 1990.

[20] S. Kermit. Cancellative medial means are arithmetic. *Duke Math J.*, 37:439–445, 1970.

[21] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, 1986.

[22] F.W. Lawvere. An elementary theory of the category of sets. *Proc. Nat. Acad. Sci. U.S.A.*, 52:1506–1511, 1964.

[23] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[24] S. Mac Lane and I. Moerdijk. *Sheaves in geometry and logic*. Springer-Verlag, New York, 1994. A first introduction to topos theory.

[25] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984.

[26] I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Ann. Pure Appl. Logic*, 104(1-3):189–218, 2000.

[27] D. Pataraia. A constructive proof of Tarski's fixed-point theorem for dcpo's. Presented in the 65th Peripatetic Seminar on Sheaves and Logic, in Aarhus, Denmark, November 1997.

[28] L.C. Paulson. *ML for the working programmer*. Cambridge University Press, Cambridge, 1991.

[29] D. Pavlović and V. Pratt. On coalgebra of real numbers. *Electronic Notes in Theoretical Computer Science*, 19, 1999.

[30] G.F. Simmons. *Introduction to Topology and Modern Analysis*. McGraw-Hill, New York, 1963.

[31] P. Taylor. *Practical foundations of mathematics*. Cambridge University Press, Cambridge, 1999.

[32] A. Turing. On computable numbers, with an application to the Entscheindungproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. See also 43:544–546, 1936.

[33] J. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, 1990.

[34] K. Weihrauch. *Computable analysis*. Springer-Verlag, 2000.

[35] E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980.

Invited Talk

Logician in the land of OS: Abstract State Machines in Microsoft

Yuri Gurevich
Microsoft Research
http://research.microsoft.com/~gurevich

Abstract

Analysis of foundational problems like "What is computation?" leads to a sketch of the paradigm of abstract state machines (ASMs). This is followed by a brief discussion on ASMs applications. Then we present some theoretical problems that bridge between the traditional LICS themes and abstract state machines.

1 Introduction

This talk was prompted by Joe Halpern's invitation letter: "My hope this year is that the invited talks will showcase the relevance of logic to the rest of CS. It seems that some discussion of abstract state machines (and their potential impact on Microsoft) would be a great theme ..."

I always had a taste for foundational questions. That is why I went to logic (from algebra) in the first place. In 1982 Michigan hired me, a logician, on the promise to become a computer scientist. Contrary to mathematical logic where the foundational questions had been more or less settled, the foundational questions of computer science were wide open. What is it that we study in computer science? What is computation? What are the peculiar dynamic systems of computer science? Thinking about these questions, I arrived at the notion of abstract state machine (ASM) as a formalization of the notion of computer system at any given level of abstraction.

The operational approach of ASMs went against the pure declarative fashion of the formal methods of the time. Many formal-methods experts still think that any operational approach is necessarily low-level and that an executable specification is a contradiction in terms. But ASMs were successful in applications. The ASM community grew and with it grew the diversity of applications; see the ASM academic website [23] where you will find in particular a bibliography [13] and Egon Börger's surveys [11, 12]. While much of ASM activity takes place in academia, it is not confined to academia. Good ASM work has been done in Siemens. There is an active ASM group in Microsoft. There are even two small ASM-based start-ups, http://www.modeled-computation.com and http://www.montages.com/.

The rest of this talk is organized as follows.

Section 2 A version of our original analysis of the fundamental questions mentioned above.

Section 3 A sketch of the ASM paradigm.

Section 4 A few words on what ASMs are good for.

Section 5 A few words on our Microsoft experience.

Section 6 Some theoretical problems related to ASMs.

Section 7 Postlude.

I showed a draft of this talk to my former student Quisani which resulted in some Q & A inserted in the text.

Acknowledgment

I am grateful to Andreas Blass, Mike Barnett, Uwe Glässer, Nikolai Tillmann and Margus Veanes for comments on this article which was written a little too quickly.

2 What is computation?

A computation can be defined as a run of a computer system. The notion of computer system should be general enough to account for future computer systems and for more abstract computations that you encounter, e.g., in the specification stage of software development. We proceed to make our notion of computer system a little more precise

2.1 Levels of abstraction

A computer system has a hierarchy of levels of abstraction. For example, you can view the execution of a C program on the level of the source program or on the level of

0-7695-1281-X/01 $10.00 © 2001 IEEE

the executable code. These are two different abstraction levels. Here we are interested in computations of a computer system with a fixed level of abstraction.

The need to fix a particular level of detail is well understood in software engineering. To this end, for example, APIs (application programming interfaces) enable the programmer to give precise syntactic information about a component — method names, typing information, etc. Typically the intended semantics is only hinted at. (And so you may want to use ASMs to fill in the gap.)

2.2 The program

A computer system is governed by a fixed program. Human society for example is not a computer system. The more focused theory of computer systems should be deeper than General System Theory.

A programmed system does not have to be closed. It can be highly interactive.

> **Q:** Is Internet a computer system in your sense?
>
> **A:** I guess this depends on the chosen level of abstraction. Even a complex system, like Internet, can be algorithmic on some levels of abstraction.
>
> **Q:** Shouldn't this apply to human society as well?
>
> **A:** You are right; it should.
>
> **Q:** Suppose that my program has loaded a bunch of classes from some library. Does this change the program of my computer system?
>
> **A:** Not necessarily. Again, this depends on the chosen level of abstraction. One possible view is this. Loading new classes changes only a part of your state; in particular the set of methods available to your program. The methods themselves can be seen as part of the active environment.
>
> **Q:** Maybe you should say "algorithmic system" rather than "computer system".
>
> **A:** Maybe. I used to say "algorithm" instead of "computer system" but there is a tendency to interpret the term "algorithm" too narrowly. Let's stick to the term "computer system" for the time being.
>
> **Q:** There are so-called non-von-Neumann systems which change their programs as they run.
>
> **A:** I saw some of them. Here is my understanding of how they work. There are fixed rules how to change the alleged program. Those rules constitute the real program. The alleged program is data.

2.3 The state

In general, a computer system is a dynamic system; it has a state that evolves in time.

> **Q:** Can a computer system be static? If yes, does it still have a state?
>
> **A:** Yes, and yes. Consider a sorting algorithm at the abstraction level where you abstract from everything except the input-output function that takes a given sequence to the sorted one. At that level of abstraction, no dynamics remains; the system still has a state (including the sorting function) but the state does not evolve in time.

2.4 So what is computation?

Computation is evolution of the state.

> **Q:** I guess you are talking about computations of a computer system at a fixed level of abstraction.
>
> **A:** Yes, I am.
>
> **Q:** This definition is not a mathematical definition.
>
> **A:** Right. It is a philosophical speculation.
>
> **Q:** I am skeptical about philosophical speculations. Give me one example of a philosophical speculation that proved to be useful.
>
> **A:** Turing's speculative proof of his thesis [27].

3 The ASM paradigm

The notion of abstract state machine (ASM) formalizes our notion of computer system given at a fixed abstraction level.

The ASM Thesis Let A be any computer system at a fixed level of abstraction. There exists an abstract state machine B that simulates A step-for-step.

> **Q:** How is this thesis different from Turing's thesis?
>
> **A:** In many ways. In particular, a Turing machine would simulate A on the level of single bits while an ASM simulates A on the given abstraction level.

The "step-for-step" requirement is crucial. In distributed computing, typically only single steps are guaranteed not to be interrupted by other agents. If B simulates A step-for-step then it can substitute for A in distributed situations. Even if B makes only two steps to simulate one step of A, some other agent can intervene between the steps of B and mess up the simulation.

In [20], we proved the thesis for the case of sequential algorithms, more exactly for sequential-time algorithms with uniformly bounded parallelism.

Q: Is it a mathematical proof or another philosophical speculation?

A: It is a mathematical proof.

Q: How can you prove a thesis? The notion of sequential algorithms is informal.

A: We formalize the notion of sequential algorithms by means of three postulates: the Sequential-Time Postulate, the Abstract-State Postulate, and the Bounded-Exploration (that is stepwise uniformly bounded exploration) Postulate.

Work on more general versions of the thesis is in progress. Instead of defining ASMs here, we just sketch the ASM paradigm. The standard reference for the ASM syntax still is [19]; a new guide is in preparation.

Let A be a computer system at a fixed level of abstraction.

3.1 States as structures

States of A are first-order structures.

Q: Why first-order? Why not second-order or higher-order?

A: Second-order and higher-order and other kinds of logical structures can be viewed as special first-order structures. See for example article [10] where weak higher-order structures are treated as first-order structures.

Q: Why should it be any kind of logic structure?

A: The vast experience in applications of mathematical logic seems to confirm that any static mathematical reality can be adequately described as first-order structure.

Q: It can be, I guess, adequately described in arithmetic.

A: Arithmetization requires excessive encoding while structure representation is virtually free from encoding.

All states of A have the same vocabulary. The vocabulary reflects the invariant aspects of the algorithm. Further the base set of the state does not change during the evolution.

Q: Many graph algorithms acquire new nodes as they run.

A: But where do they take those new nodes from? We assume that the initial state has an infinite reserve of elements to be used as nodes or whatever. A special `import` (called also `create`) operator is used to fish out elements from the reserve and bring them to the foreground.

The set of states of A is closed under isomorphisms. Intuitively, isomorphic structures are representations of the same state. The details of representation should not matter.

Q: If computation is state evolution and states are structures then computation is structure evolution.

A: That is why abstract state machines used to be called evolving structures or evolving algebras.

Q: Why algebras?

A: An algebra is a structure whose vocabulary consists of function symbols. In logic, relations are different from functions because their values live outside the structure. We tweaked the definition of first-order structures so that the Boolean values are always inside and thus our states are algebras.

3.2 State as a memory

In logic or algebra, structures are static. Our structures are dynamic. A state X is a memory (or store). If f is a function symbol of arity j in the vocabulary of X and if $\bar{a}$ is a j-tuple of elements of X then the pair $(f, \bar{a})$ is a *location* of X. The *content* of that location is the element $f(\bar{a})$.

3.3 Actions

An *atomic update* of a state X changes the content of one location of X. Since the vocabulary of A is fixed and the base set of the state does not change during the evolution, the set of locations does not change either. It follows that any transition from one state to another is characterized by an *update set*, a set of atomic updates.

The ASM syntax provides means to program atomic updates as well as various update sets. For example, if ϕ is a Boolean-valued term and R is an ASM rule generating an update set U at a state X then the rule `if` ϕ `then` R generates either U or $\emptyset$ over X depending on whether ϕ evaluates to `true` or to `false` over X.

Q: I guess state changes should respect isomorphisms of structures.

A: Of course. In [20], this is a part of the abstract-state postulate.

3.4 Runs

You have in general a number of computing agents executing their programs. It is convenient to think in terms of a global state. A move by an agent changes only a finite set of locations of the global state. Concurrent moves of different agents produce consistent changes. A run is a partial order of moves of various agents.

Q: Your global state is some kind of shared memory.

A: It is not a conventional shared memory.

Q: Consider a distributed system, say a network of computers. To make it more interesting, let us assume that different computers are located on different planets so that, by the relativity theory, the whole system does not have a global time. The computers exchange information via messages. Are there meaningful states of the system?

A: Yes, they are mathematical abstractions [19].

Further, agents themselves are represented in the state. The computation can destroy agents and create new ones. There could be various relations and functions involving agents [19].

3.5 ASMs and set theory

In a 1993 Dagstuhl conference, Andreas Blass said the following about formalizing algorithms as ASMs: "after a while it becomes clear that any 'reasonable' algorithm can be written as an ASM, just as any 'reasonable' proof can be formalized in ZFC." This observation is analyzed and developed further in the chapter "ASMs and Set Theory" of his article "Abstract State Machines and Pure Mathematics" [4].

4 What are ASMs good for

The most obvious use of ASMs is to write executable specifications. Here is a sorting example.

You don't need ASMs to specify that a sorting algorithm should sort. But suppose that, for some reason, e.g. security, you need that your sorting is in-place so that you only swap elements of the given array. Suppose further that you can do only one swap at a time. There are numerous ways to implement such sorting: quicksort, bubble sort, etc. Here is an ASM spec of in-place one-swap-a-time sorting. Suppose that a is an array with the set I of indices.

```
choose i,j in I with i<j and a[i]>a[j]
   do in-parallel
      a[i]:=a[j]
      a[j]:=a[i]
```

This rule is supposed to be executed over and over again until the computation halts (when the choice set becomes empty). This is the most general in-place one-swap-a-time sorting (such that every swap makes the array more sorted). You can employ various choice strategies and thus get more refined sorting algorithms; a refinement like quicksort is much more efficient than the spec. But the spec is executable as is, and appropriate ASM tools can execute it.

Q: Your notion of specification is very broad.

A: Yes. Whenever you have a pair of algorithms A and B so that B refines A, A is a spec for B. This includes the case when A is static and so the spec is declarative.

Q: Why is it important that specifications are executable?

A: Imagine that you have designed a cool product with many interesting features. Developers code it; this may take a while. Eventually testers may discover that the design was flawed and needs to be changed. You wish you could have played with your design before coding.

There are many more kinds of applications of ASMs; see [23] where you will find in particular a bibliography [13] and Egon Börger's surveys [11, 12].

5 ASMs in Microsoft

Jim Kajiya at Microsoft Research realized the potential of ASMs. In late summer of 1998, he invited me to start a new group, and I accepted. The ASM project had become more and more engineering, and I could use help. In addition, I was tired of analyzing old software and excited about the possibility to participate in the development of new software.

The new group was called Foundations of Software Engineering (FSE). By now we have a strong and busy ASM team that never seems to find time to dress up its outside window [14]. Our first priority is to develop a good tool to write and execute ASMs. A number of such tools have been developed in academia; see [23]. Two of these tools, ASM Workbench and ASM Gopher, have been successfully used at Siemens. However none of the tools was a good fit for the software development environment of Microsoft, and in particular for COM, Microsoft's Component Object Model [24]. We had to start from scratch.

Q: What is COM?

A: I quote from [2]: "Microsoft software is usually composed of COM components. These are really just static containers of methods. In your PC, you will find dynamic-link libraries (DLLs); a

library contains one or more components (in compiled form). COM is a language-independent as well as machine-independent binary standard for component communication. An API for a COM component is composed of *interfaces*; an interface is an access point through which one accesses a set of methods. A client of a COM component never accesses directly the component's inner state, or even cares about its identity; it only makes use of the functionality provided by different methods behind the interface (or by requesting a different interface)."

The tool development in the group is headed by Wolfram Schulte, my first hire, who came to Microsoft in the summer of 1999 from the University of Ulm in Germany after completing his ASM-related habilitation there. Our main tool is called AsmL (ASM Language). It is an executable-specification language.

Q: What does it mean? Another high-level programming language?

A: It is a high-level programming language that implements the ASM paradigm. Accordingly it is highly parallel.

Q: What about that COM?

A: AsmL is COM compliant. You can specify a component, and the spec will have full COM connectivity. For example, a spec of a debugger may be much more concise and abstract than a real debugger, but it will be treated as a debugger by other COM components.

Q: Is AsmL optimized for efficiency or expressivity?

A: It is a pragmatic compromise but typically expressivity comes first.

Q: Are there product groups within Microsoft that use ASM technology?

A: Yes.

Q: Name one.

A: Universal Plug and Play.

This seems to be a wrong place to go into the details of our work. (A bunch of our papers should appear later this year in the Proceedings of ASM'2001 in Springer Lecture Notes in Computer Science. A few additional papers are headed elsewhere. We'll try to keep the website [14] current.) Instead let me share a few lessons that the group learned during its short existence.

- Verification isn't everything. Verification is great ... when it is feasible. A spec is a basis not only for verification but also for testing, documentation, etc. Partial improvements can have a big impact
- Stay relevant. A spec must be testable and up-to-date.
- Integration is crucial. Without integration your tool may be useless. Integrate with the relevant developer environment (in our case, it is Microsoft Visual Studio). Integrate with the relevant run-time environments (in our case, they are COM, .NET and various libraries).

6 On ASM-related theoretical problems

I was asked more than once about ASM-related theoretical problems. Many appetizing foundational problems arise in applications. For example, what are objects and classes [21]? But let me keep closer to more traditional LICS themes (with hope to bridge between those themes and ASMs).

6.1 Fine complexity classes

The notion of polynomial time is very robust. The usual computation models including the Turing model give the same notion of polynomial time. In [22], we show that the usual computation models other than the Turing model give the same notion of nearly linear (that is linear times polylog) time. Linear time is much more sensitive to the choice of computation model, and there are numerous versions of linear time in use. One example is the linear time of computational geometry. The ASM model may have enough parameters to take care of all these versions of linear time — maybe. I did not investigate this.

In [6], we proved the linear-time hierarchy theorem for ASMs (that asserts that, as c varies, the classes of functions computable in time $c \cdot n$ form a proper hierarchy). As we wrote there, "One long-term goal of this line of research is to prove linear lower bounds for linear time problems".

If you work with linear time and consider simulations, it is natural to require that simulation is lock-step, that is there exists a fixed k such that the simulator spends at most k steps to simulate one step of the simulatee. In [6], we used lock-step simulations with preprocessing to construct a diagonalizing machine and thereby proved the linear-time hierarchy theorem. Lock-step simulation deserves to be studied in its own right. To this end, Andreas Blass constructed a more involved diagonalizing machine that avoids preprocessing (unpublished).

It seems that the study of fine complexity classes was held back by the absence of an appropriate computation model. We hope that ASM can serve as such a model.

6.2 Computations with abstract structures

Contrary to conventional computation models, like Turing machines or random access machines, ASMs accept abstract structures as inputs. For example, an input could be a graph rather than a string (or adjacency matrix) representation of the graph.

> **Q:** Why is this important? Real computers do not accept abstract structures as inputs.
>
> **A:** You routinely abstract from representation details when you do specifications. But such abstraction is not confined to specifications. Suppose for example that I have a database in my computer and I ship it to you. You store it in your computer but the representation of the database in your computer will surely differ from that in mine. A query to database should not depend on the representation. To this end, popular query languages abstract from the representation, so that abstract databases are treated as inputs to queries. This is an important issue in database theory and practice [1].

In [18], I conjectured that there is no logic (or computation model) for polynomial-time computations with abstract structures; the conjecture implies P≠NP and remains open.

> **Q:** I do not understand your conjecture. How can you quantify over logics?
>
> **A:** I assume that every logic satisfies some minimal requirements, in particular that the set of well-formed formulas is recursive.

In [10], ASMs were used as a computation model (and logic of a sort, called BGS) for a rich natural class of polynomial-time computations with abstract structures. Later Shelah proved the zero-one law for BGS [26, 5]. In particular, we show in [10] that counting is not available in BGS and that BGS cannot decide whether a bipartite graph admits a perfect matching. Later it was shown in [9] that if one adds counting to BGS then the perfect matching problem for bipartite graphs becomes expressible. It remains open whether the perfect matching problem for arbitrary graphs is expressible in BGS with counting. It is also open whether BGS with counting captures polynomial time. Other specific problems along these lines are discussed in [9]. In [16], ASMs were used to study logspace computations with abstract structures.

The complexity theory of computations with abstract structures deserves to be developed further.

6.3 Metafinite models

In [17], I preached finite model theory because many structures naturally arising in computer science are finite. In particular (the states of) relational databases are finite. But are they really finite in all cases? A database may use real numbers for example; where do those numbers "live"? Now consider the world of, say, the C programming language. It has arrays, records, arrays of records, records of arrays, and so on. It is convenient to model states of computer systems as infinite structures where only a finite part is active. To this end, we defined and studied metafinite structures in [15]. A metafinite structure has a finite primary part and possibly infinite secondary part.

In the case of a program state, the primary part reflects the active foreground and the infinite part reflects the passive background. One example is [10] where the background is the collection of hereditarily finite sets over the elements of the input structure. In general, background structures contain all the material (like maps of sets of sequences of maps) that the program may need. The notion of background was formalized in [7].

Metafinite structures are really ubiquitous and deserve more attention.

6.4 Interesting logics

What are logics appropriate to metafinite structures? That question has been addressed in [15]. Basically, you can quantify over the primary part only. The secondary part may have powerful operations. In the case of reals, for example, you may have multiset operations like sum, product, average, median. But you can't quantify over the secondary part.

The choice operator of ASMs (illustrated above, in Section 4) is typical for computer science. It is an independent-choice operator: different invocations of it produce independent choices. It differs from the epsilon operator of Hilbert, the classical choice operator of mathematical logic; different invocations of the epsilon operator over the same set produce the same result. In [8], we investigated the logic of the ASM choice operator. We found that this fascinating logic is much weaker than the logic of the epsilon operator.

There are other ASM-related logics waiting to be investigated. One example is first-order logic with `undef`, a special element that allows you to turn partial functions into total ones. This `undef` is different from `diverges` of recursive-function theory. An ASM program can refer to `undef` explicitly; in particular we allow tests like `x = undef`, and the equality `undef = undef` holds. This explicit use of `undef` makes the logic of `undef` more powerful than the other first-order logics of partial functions that I am aware of.

Until now we spoke about static logics. Once one introduces state transitions, new challenging issues appear.

What is the logic of the import operator mentioned above in Section 3? Unlike the choice operator, the import operator produces a different element every time it is invoked.

In the sequential-time case, an ASM program describes a single step (to be iterated). The state changes only at the end of the step, not at the middle. There are no side effects during the execution of one step. This feature of the ASM paradigm should allow one to develop clean logics to reason about at least one step of the program.

One may want also to use automated and partially automated systems, including model checking systems, to reason about the behavior of abstract state machines. The ASM community has some experience in this direction; see [25, 3] and the section on Mechanical Verification in [23]. We have a long way to go though.

7 Postlude

Logic that we use and apply in computer science is mathematical logic developed originally to build foundations of mathematics and to solve the problems in foundations of mathematics that arose in the beginning of twentieth century. Logicians distinguish clearly between syntax and semantics and strive to clarify both syntactical and semantical issues. Computer science applications of logic are much different from mathematical applications. Some of the strongest methods of mathematical logics, like the priority method and forcing, have not found direct applications in computer science. But the foundational tradition of logic is of great value to computer science at this stage of its development.

But computer science is not a purely mathematical discipline. It is an engineering discipline as well. In applications, it does not suffice to prove that the problem is decidable or even polynomial-time decidable. You may need a program that works reasonably fast on real computers. Some engineering compromises have to be made. It is not only syntax and semantics that we should worry about. It is also pragmatics. It may mess up your clean constructions, but it may also enhance them and make them work for the benefit of many.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A Case Study. In Y. Gurevich et al., editors, *Abstract State Machines: Theory and Applications (Proceedings of ASM'2000)*, volume 1912 of *LNCS*, pages 367–379. Springer-Verlag, 2000.

[3] D. Beauquier and A. Slissenko. A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class. *Annals of Pure and Applied Logic*. to appear.

[4] A. Blass. Abstract State Machines and Pure Mathematics. In Y. Gurevich et al., editors, *Abstract State Machines: Theory and Applications (Proceedings of ASM'2000)*, volume 1912 of *LNCS*, pages 9–21. Springer-Verlag, 2000.

[5] A. Blass and Y. Gurevich. Strong Extension Axioms and Shelah's Zero-One Law for Choiceless Polynomial Time. to appear.

[6] A. Blass and Y. Gurevich. The Linear Time Hierarchy Theorem for RAMs and Abstract State Machines. *Journal of Universal Computer Science*, 3(4):247–278, 1997.

[7] A. Blass and Y. Gurevich. Background, Reserve, and Gandy Machines. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL'2000*, volume 1862 of *LNCS*, pages 1–17. Springer-Verlag, 2000.

[8] A. Blass and Y. Gurevich. Logic of Choice. *Journal of Symbolic Logic*, 65(3):1264–1310, 2000.

[9] A. Blass, Y. Gurevich, and S. Shelah. On Polynomial Time Computation Over Unordered Structures. to appear.

[10] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. *Annals of Pure and Applied Logic*, 100(1–3):141–187, 1999.

[11] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer, 1995.

[12] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Proceedings of FM-Trends'98, Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer, 1999.

[13] E. Börger and J. K. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of European Association for Theoretical Computer Science*, 1998. Number 64, February 1998, pp. 105–128.

[14] Foundations of Software Engineering Group at Microsoft Research. `http://research.microsoft.com/fse`.

[15] E. Grädel and Y. Gurevich. Metafinite Model Theory. *Information and Computation*, 140(1):26–81, 1998.

[16] E. Grädel and M. Spielmann. Logspace Reducibility via Abstract State Machines. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99*, volume 1709 of *LNCS*, pages 1738–1757. Springer-Verlag, 1999.

[17] Y. Gurevich. Toward logic tailored for computational complexity. In M. Richter et al, editors, *Computation and Proof Theory: Logic Colloquium 1983*, volume 1104 of *LNCS*, pages 175–216. Springer-Verlag, 1984.

[18] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[19] Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[20] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[21] Y. Gurevich, W. Schulte, and M. Veanes. A Richer ASM Language (tentative title). In E. Börger and U. Glässer, editors, *Proceedings of ASM'2001*, LNCS. Springer-Verlag, 2001. to appear.

[22] Y. Gurevich and S. Shelah. Nearly linear time. In *Symposium on Logical Foundations of Computer Science*, volume 363 of *LNCS*, pages 108–118, Springer-Verlag, 1989.

[23] J. K. Huggins. Michigan Webpage on Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.

[24] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[25] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[26] S. Shelah. Choiceless Polynomial Time Logic: Inability to Express. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL'2000*, volume 1862 of *LNCS*, pages 72–125. Springer-Verlag, 2000.

[27] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of London Mathematical Society (2)*, 42:230–236, 1936–37. Correction, *ibid.* 43, 544–546.

Session 4

Eliminating definitions and Skolem functions in first-order logic

Jeremy Avigad
Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In any classical first-order theory that proves the existence of at least two elements, one can eliminate definitions with a polynomial bound on the increase in proof length. In any classical first-order theory strong enough to code finite functions, including sequential theories, one can also eliminate Skolem functions with a polynomial bound on the increase in proof length.

1 Introduction

When working with a first-order theory, it is often convenient to use definitions. That is, if $\varphi(\vec{x})$ is a first-order formula with the free variables shown, one can introduce a new relation symbol R to abbreviate φ, with defining axiom $\forall \vec{x}\ (R(\vec{x}) \leftrightarrow \varphi(\vec{x}))$. Of course, this definition can later be eliminated from a proof, simply by replacing every instance of R by φ. But suppose the proof involves nested definitions, with a sequence of relation symbols $R_0, \ldots, R_k$ abbreviating formulae $\varphi_0, \ldots, \varphi_k$, where each φ_i may have multiple occurrences of $R_0, \ldots, R_{i-1}$. In that case, the naive elimination procedure described above can yield an exponential increase in the length of the proof.

In Section 2, I show that if the underlying theory proves that there are at least two elements in the universe, a more careful translation allows one to eliminate the new definitions with at most a polynomial increase in length. The proof is not difficult, but it relies on the assumption that equality is included in the logic. A similar trick has been used by Solovay in simulating iterated definitions efficiently, as discussed in [11, Section 3.2]. Consequently, the result proved here may be folklore, but to my knowledge it has not appeared in the literature, and it is needed in Section 3.

It is also sometimes convenient, in a first-order setting, to introduce Skolem functions. If $\varphi(\vec{x}, y)$ is any formula with the free variables shown and f is a new function symbol, one can add an axiom, $\forall \vec{x}, y\ (\varphi(\vec{x}, y) \rightarrow \varphi(\vec{x}, f(\vec{x}))$, asserting, in words, "if any y satisfies $\varphi(\vec{x}, y)$, $f(\vec{x})$ does." There is an easy model-theoretic proof of the fact that this does not alter the set of consequences in the original language: any first-order model of the original theory can be expanded to a model where f denotes such a choice function. Explicit syntactic proofs of this fact are, however, somewhat more difficult. The first such proof appears in Hilbert and Bernays' *Grundlagen der Mathematik* [8], using the epsilon substitution method; a proof by Maehara using cut-elimination is discussed in [14]; and another proof due to Shoenfield is found in [13] (see also the discussion in [12]). All these procedures are, unfortunately, worse than exponential.

In Section 3, I show that if the underlying theory allows for a modicum of coding, one can also eliminate Skolem functions with at most a polynomial increase in proof length. The idea is to use an internal, iterated forcing argument to add the new functions. The forcing conditions involved are finite approximations to the Skolem functions being added, so the constraint on the underlying theory is that it provides an adequate representation of finite functions. The specific requirements are spelled out below; any sequential theory of arithmetic meets these criteria. While forcing methods have been used to establish lower bounds in proof complexity (see [1, 9, 10]), here they are used to establish upper bounds; similar forcing arguments can be found in [2, 3, 4, 5].

The question as to whether or not definitions can be eliminated efficiently from propositional proof systems is a major open question in the field of proof complexity. The results here show that the answer is "yes" for most *first-order* proof systems, though the most general statement of the problem is equivalent to the propositional version. Issues related to Skolem functions are similarly important to computer science, since most automated search procedures use Skolemization in one form or another. The question as to the increase in proof length when eliminating a sin-

0-7695-1281-X/01 $10.00 © 2001 IEEE

gle Skolem function from a proof in pure first-order logic is listed as open problem 22 in [6]. Once again, though the results here do not settle the most general statement of the problem, they show that for many natural theories such an efficient elimination is possible. In Section 4, I discuss some questions that remain.

2 Eliminating definitions

If d is a proof of a sentence ψ from a set of axioms Γ in first-order logic, then $|d|$ denotes the length of d, according to the number of symbols. Krajíček [9] and Pudlák [11] provide good general references on the lengths of proofs.

In this section and the next I will show that in certain circumstances one can eliminate definitions and/or Skolem functions from a proof d in such a way that the length of the resulting proof is bounded by a polynomial in $|d|$. In doing so, I will not make an effort to compute the exact polynomial; rather, I will repeatedly appeal to the fact that the set of polynomials in $|d|$ is closed under addition, multiplication, and composition. It will be clear from the proofs that in fact all the translations considered can be carried out in polynomial time.

By "first-order logic," I mean first-order logic with equality, in any of the standard natural deduction calculi, Hilbert-style calculi, or sequent calculi with cut described in [15]. By a theorem due to Krajíček, up to polynomial-time equivalence it does not matter whether we take proofs to be given by trees or sequences of lines (see [11, Section 4], or [9, Section 4.5] for the propositional case). In fact, the proof of Theorem 2.2 only assumes that there is a representation of $\varphi \to \psi$ which uses φ only once. If $\leftrightarrow$ is assumed to be one of the basic connectives, one can simplify the central argument somewhat; but the proof below works in either case.

I will use the following conventions: $\vec{x}$ and $\vec{t}$ denote sequences of variables and terms, respectively, and typically their lengths can be inferred from the context. Introducing a formula as $\varphi(\vec{x})$ only serves to distinguish the sequence of variables $\vec{x}$, after which $\varphi(\vec{t})$ denotes the result of simultaneously substituting $\vec{t}$ for $\vec{x}$, renaming bound variables in φ if necessary.

Definition 2.1 *Let Γ be a set of first-order sentences in a language L. Say that Γ has an* efficient elimination of definitions *if there is a polynomial $p(x)$ such that the following holds: whenever $R_0(\vec{x}_0), \ldots, R_k(\vec{x}_k)$ are new relation symbols of various arities, $\varphi_0(\vec{x}_0), \ldots, \varphi_k(\vec{x}_k)$ are formulae such that each φ_i is in the language $L \cup \{R_0, \ldots, R_{i-1}\}$, and d is a proof of a formula ψ in L from*

$$\Gamma \cup \{\forall \vec{x}_0\ (R_0(\vec{x}_0) \leftrightarrow \varphi_0(\vec{x}_0)), \ldots, \\ \forall \vec{x}_k\ (R_k(\vec{x}_k) \leftrightarrow \varphi_0(\vec{x}_k))\},$$

then there is a proof d' of ψ from Γ using only formulae in L, with $|d'| \leq p(|d|)$.

This definition is monotone in Γ: if Γ has an efficient elimination of definitions and $\Gamma' \supseteq \Gamma$ then, by the deduction theorem, Γ' has an efficient elimination of definitions as well. The main theorem in this section is the following:

Theorem 2.2 *$\{\exists x, y\ (x \neq y)\}$ has an efficient elimination of definitions.*

Proof. The proof will occupy most of this section. Let $R_0, \ldots, R_k, \varphi_0, \ldots, \varphi_k, \psi$, and d be as in the definition. We can assume that each of the defining axioms occurs at least once in the proof, since if the axiom for R_i does not occur in the proof we can replace each occurrence of R_i by an arbitrary sentence, say $\forall x\ (x = x)$. As a result, we can assume that k and $|\varphi_0|, \ldots, |\varphi_k|$ are all less than $|d|$, and so it suffices to bound the length of the final proof by a polynomial in these values.

Let a and b be new constant symbols. It suffices to find a short (i.e. polynomially bounded) proof of ψ from $\{a \neq b\}$. For, if we can find a short proof of $a \neq b \to \psi$, we can replace a and b by variables and obtain a short proof of ψ from $\exists x, y\ (x \neq y)$.

First, note that without loss of generality we can assume that all the definitions are given by prenex formulae. If the propositional connectives are among $\{\wedge, \vee, \to, \neg\}$ this is so because any formula involving these connectives can be proved equivalent to one that is prenex, with a proof whose length is bounded by a polynomial in the length of the original formula. On the other hand, if, say, $\leftrightarrow$ is a propositional connective, one can introduce additional definitions to abbreviate subformulae and ensure that all the definitions are prenex. Alternatively, one can first use definitions to eliminate $\leftrightarrow$ as in the proof of Corollary 2.5, and then proceed as before.

In the following argument, if θ is a formula with a relation symbol $R(\vec{y})$ and $\eta(\vec{y})$ is a formula with the free variables shown, it will be convenient to write $\theta[\eta/R]$ for the result of replacing each atomic formula $R(\vec{t})$ by $\eta(\vec{t})$. At other times, I will write $\theta[R(t_1, \ldots, t_m)]$ to indicate that an atomic formula $R(t_1, \ldots, t_m)$ occurs in the quantifier-free formula θ; thereafter, $\theta[\eta]$ denotes the result of replacing $R(t_1, \ldots, t_m)$ by η. While this notation is potentially problematic, the intention should always be clear from the context.

For notational convenience, we may assume that all of the relations R_i have the same arity. We will need a way of representing the numbers $0, \ldots, k$. Let $z_0, \ldots, z_k$ be a sequence of variables, write $\overline{0}$ for the sequence $a, b, b, b, \ldots$, $\overline{1}$ for the sequence $b, a, b, b, \ldots$, and, more generally, $\overline{j}$ for the sequence of length $k+1$ that has an a in the jth position and b's elsewhere.

Our strategy will be to define a sequence of formulae $\hat{\varphi}_0(\vec{z}, u, \vec{x}), \ldots \hat{\varphi}_k(\vec{z}, u, \vec{u})$, with length bounded by a polynomial in $|d|$, such that for each $i \leq k$ the following equivalences are all provable from $a \neq b$:

- $\hat{\varphi}_i(\overline{j}, a, \vec{x}) \leftrightarrow \hat{\varphi}_{i-1}(\overline{j}, a, \vec{x})$, for each $j < i$
- $\hat{\varphi}_i(\overline{j}, b, \vec{x}) \leftrightarrow \neg\hat{\varphi}_{i-1}(\overline{j}, a, \vec{x})$, for each $j < i$
- $\forall \vec{x}\ (\hat{\varphi}_i(\overline{i}, a, \vec{x}) \leftrightarrow \varphi_i(\vec{x})[\hat{\varphi}_{i-1}(\overline{0}, a, \vec{x})/R_0, \ldots, \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{x})/R_{i-1}])$
- $\forall \vec{x}\ (\hat{\varphi}_i(\overline{i}, b, \vec{x}) \leftrightarrow \neg\varphi_i(\vec{x})[\hat{\varphi}_{i-1}(\overline{0}, a, \vec{x})/R_0, \ldots, \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{x})/R_{i-1}])$.

In other words, for each i and $j \leq i$, $\hat{\varphi}_i(\overline{j}, a, \vec{x})$ is an efficient representation of R_j, and $\hat{\varphi}_i(\overline{j}, b, \vec{x})$ is an efficient representation of $\neg R_j$. The idea is to use quantifiers and equality so that only a single instance of $\hat{\varphi}_i$ is used in the definition of $\hat{\varphi}_{i+1}$. Note that the clauses above imply that for each i and $j \leq i$, we have $\hat{\varphi}_i(\overline{j}, a, \vec{x}) \leftrightarrow \neg\hat{\varphi}_i(\overline{j}, b, \vec{x})$.

The sequence $\hat{\varphi}_0, \ldots, \hat{\varphi}_k$ is defined recursively. Start by taking $\hat{\varphi}_0(\vec{z}, u, \vec{x})$ to be the formula

$$(u = a \rightarrow \varphi_0(\vec{x})) \wedge (u = b \rightarrow \neg\varphi_0(\vec{x})).$$

For $i > 0$, assuming $\hat{\varphi}_0, \ldots, \hat{\varphi}_{i-1}$ have been defined, the following shows how to determine $\hat{\varphi}_i$. Since we are assuming that all the definitions are prenex, $\varphi_i(\vec{x})$ is of the form

$$Q_1 y_1 \ldots Q_m y_m\ \tilde{\varphi}[R_0(\vec{t}_{0,0}), \ldots, R_0(\vec{t}_{0,l_0}), \ldots, R_{i-1}(\vec{t}_{i-1,0}), \ldots, R_{i-1}(\vec{t}_{i-1,l_{i-1}})],$$

where $\tilde{\varphi}$ is quantifier-free and the sequence in square brackets shows all instances of atomic formulae in $\tilde{\varphi}$ involving $R_0, \ldots, R_{i-1}$. In general, the sequences of terms $\vec{t}_{j,p}$ depend on the quantified variables $y_1, \ldots, y_m$ as well as the free variables $\vec{x}$ of φ_i, but I will not display these variables explicitly. Our task is to write down a formula $\hat{\varphi}_i(\vec{z}, u, \vec{x})$ such that

1. for each $j < i$, $\varphi_i(\overline{j}, a, \vec{x})$ is equivalent to $\varphi_{i-1}(\overline{j}, a, \vec{x})$;
2. for each $j < i$, $\varphi_i(\overline{j}, b, \vec{x})$ is equivalent to $\neg\varphi_{i-1}(\overline{j}, a, \vec{x})$;
3. $\hat{\varphi}_i(\overline{i}, a, \vec{x})$ is equivalent to the displayed formula above, with each $R_j(\vec{t}_{j,p})$ replaced by $\hat{\varphi}_{i-1}(\overline{j}, a, \vec{t}_{j,p})$;
4. $\hat{\varphi}_i(\overline{i}, b, \vec{x})$ is equivalent to the negation of the formula just described; and
5. in the definition of $\hat{\varphi}_i$, $\hat{\varphi}_{i-1}$ is used only once.

In order to do 3 and 4 simultaneously, we need duplicate copies of some of the variables and terms. Let $Q'_1, \ldots, Q'_m$ denote the quantifiers dual to $Q_1, \ldots, Q_m$. Pick a new sequence of variables $y'_1, \ldots, y'_m$, and let

$$\vec{t}_{0,0}, \ldots, \vec{t}_{0,l_0}, \ldots, \vec{t}_{i-1,0}, \ldots, \vec{t}_{i-1,l_{i-1}}$$

denote the sequences of terms obtained by replacing the $y_1, \ldots, y_m$ by $y'_1, \ldots, y'_m$ in each $\vec{t}_{j,p}$. Finally, let

$$v_{0,0}, \ldots, v_{0,l_0}, \ldots, v_{i-1,0}, \ldots, v_{i-1,l_{i-1}}$$
$$v'_{0,0}, \ldots, v'_{0,l_0}, \ldots, v'_{i-1,0}, \ldots, v'_{i-1,l_{i-1}}$$
$$v''_0, \ldots, v''_{i-1}$$

be sequences of new variables. We will use the variables $v_{j,p}$ to represent the truth values of $\hat{\varphi}_{i-1}(\overline{j}, a, \vec{t}_{j,p})$, the variables $v'_{j,p}$ to represent the truth values of $\hat{\varphi}_{i-1}(\overline{j}, a, \vec{t}'_{j,p})$, and the variables v''_j to represent the truth values of $\hat{\varphi}_{i-i}(\overline{j}, a, \vec{x})$, where the "truth value" is a if the corresponding formula is true, and b if it is false.

The formula $\hat{\varphi}_i(\vec{z}, u, \vec{x})$ is defined to be

$$\begin{aligned}
&Q_1 y_1 \ldots Q_m y_m\ Q'_1 y'_1 \ldots Q'_m y'_m\ \forall \vec{v}, \vec{v}', \vec{v}'' \\
&\Big(Eval(\vec{v}, \vec{v}', \vec{v}'') \rightarrow \\
&\quad \bigwedge_{j<i} (\vec{z} = \overline{j} \wedge u = a \rightarrow v''_j = a) \wedge \\
&\quad \bigwedge_{j<i} (\vec{z} = \overline{j} \wedge u = b \rightarrow v''_j \neq a) \wedge \\
&\quad (\vec{z} = \overline{i} \wedge u = a \rightarrow \tilde{\varphi}[v_{0,0} = a, \ldots, v_{0,l_0} = a, \\
&\qquad \ldots, v_{i-1,0} = a, \ldots, v_{i-1,l_{i-1}} = a]) \wedge \\
&\quad (\vec{z} = \overline{i} \wedge u = b \rightarrow \neg\tilde{\varphi}[v'_{0,0} = a, \ldots, v'_{0,l_0} = a, \\
&\qquad \ldots, v'_{i-1,0} = a, \ldots, v'_{i-1,l_{i-1}} = a])\Big)
\end{aligned}$$

where $Eval(\vec{v}, \vec{v}', \vec{v}'')$ is the formula

$$\begin{aligned}
&\forall \vec{r}\, \forall s \in \{a, b\}\ \forall \vec{w}\ \Big(\hat{\varphi}_{i-1}(\vec{r}, s, \vec{w}) \rightarrow \\
&\quad \bigwedge_{j<i} (\vec{r} = \overline{j} \wedge \vec{w} = \vec{x} \rightarrow v''_j = s) \wedge \\
&\quad \bigwedge_{j<i} \bigwedge_{p<l_j} (\vec{r} = \overline{j} \wedge \vec{w} = \vec{t}_{j,p} \rightarrow v_{j,p} = s) \wedge \\
&\quad \bigwedge_{j<i} \bigwedge_{p<l_j} (\vec{r} = \overline{j} \wedge \vec{w} = \vec{t}'_{j,p} \rightarrow v'_{j,p} = s)\Big).
\end{aligned}$$

Here $\forall s \in \{a, b\}\ \theta$ abbreviates $\forall s\ (s = a \vee s = b \rightarrow \theta)$. Note that $Eval(\vec{v}, \vec{v}', \vec{v}'')$ also depends on the free variables $\vec{x}, \vec{y}, \vec{y}'$ (because the terms $t_{j,p}$ and $t'_{j,p}$ do), but I will continue to leave these variables implicit.

First, let us check that each $\hat{\varphi}_i(\vec{x}_i, u)$ satisfies the right equivalences, and then let us worry about the length. Inductively we know, for each $j \leq i - 1$, that

$$\forall \vec{x}\ (\hat{\varphi}_{i-1}(\overline{j}, a, \vec{x}) \leftrightarrow \neg\hat{\varphi}_{i-1}(\overline{j}, b, \vec{x}))$$

is provable from $a \neq b$. We can use this to show

$$\forall \vec{x}, \vec{y}, \vec{y}' \; \exists \vec{v}, \vec{v}', \vec{v}'' \; Eval(\vec{v}, \vec{v}', \vec{v}'')$$

as well as

$$\begin{aligned}
\forall \vec{x}, \vec{y}, \vec{y}', \vec{v}, \vec{v}', \vec{v}'' \; \Big(& Eval(\vec{v}, \vec{v}', \vec{v}'') \rightarrow \\
& \bigwedge_{j<i} (v''_j = a \leftrightarrow \hat{\varphi}_{i-1}(\bar{j}, a, \vec{x})) \wedge \\
& \bigwedge_{j<i} \bigwedge_{p<l_j} (v_{j,p} = a \leftrightarrow \hat{\varphi}_{i-1}(\bar{j}, a, \vec{t}_{j,p})) \wedge \\
& \bigwedge_{j<i} \bigwedge_{p<l_j} (v'_{j,p} = a \leftrightarrow \hat{\varphi}_{i-1}(\bar{j}, a, \vec{t}_{j,p})) \Big).
\end{aligned}$$

But then, going back to the definition of $\hat{\varphi}_i$, we see that for $j < i$, $\hat{\varphi}_i(\bar{j}, a, \vec{x})$ is equivalent to $\hat{\varphi}_{i-1}(\bar{j}, a, \vec{x})$, and $\hat{\varphi}_i(\bar{j}, b, \vec{x})$ is equivalent to $\neg\hat{\varphi}_{i-1}(\bar{j}, a, \vec{x})$. Also, $\hat{\varphi}_i(\bar{i}, a, \vec{x})$ is equivalent to

$$\begin{aligned}
& Q_1 y_1 \ldots Q_m y_m \varphi[\hat{\varphi}_{i-1}(\bar{0}, a, \vec{t}_{0,0}), \ldots, \\
& \quad \hat{\varphi}_{i-1}(\bar{0}, a, \vec{t}_{0,l_0}), \ldots, \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{t}_{i-1,0}), \\
& \quad \ldots, \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{t}_{i-1,l_{i-1}})]
\end{aligned}$$

and so we have

$$\begin{aligned}
\hat{\varphi}_i(\bar{i}, a, \vec{x}) \leftrightarrow \varphi_i(\vec{x})[& \hat{\varphi}_{i-1}(\bar{0}, a, \vec{x})/R_0, \ldots, \\
& \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{x})/R_{i-1}];
\end{aligned}$$

and $\hat{\varphi}_i(\bar{i}, b, \vec{x})$ is equivalent to

$$\begin{aligned}
& Q'_1 y'_1 \ldots Q'_m y'_m \neg\varphi[\hat{\varphi}_{i-1}(\bar{0}, a, \vec{t}_{0,0}), \ldots, \\
& \quad \hat{\varphi}_{i-1}(\bar{0}, a, \vec{t}_{0,l_0}), \ldots, \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{t}_{i-1,0}), \\
& \quad \ldots, \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{t}_{i-1,l_{i-1}})]
\end{aligned}$$

and so we have

$$\begin{aligned}
\hat{\varphi}_i(\bar{i}, b, \vec{x}) \leftrightarrow \neg\varphi_i(\vec{x})[& \hat{\varphi}_{i-1}(\bar{0}, a, \vec{x})/R_0, \ldots, \\
& \hat{\varphi}_{i-1}(\overline{i-1}, a, \vec{x})/R_{i-1}],
\end{aligned}$$

as required.

As far as length is concerned, it is not hard to check that the number of symbols occurring in $\hat{\varphi}_i$ apart from the instance of $\hat{\varphi}_{i-1}$ can be bounded by a polynomial in $|d|$ (in fact, even a linear one). In other words, there is a polynomial p such that for each i we have $|\hat{\varphi}_i| \leq p(|d|) + |\hat{\varphi}_{i-1}|$, and hence $|\hat{\varphi}_i| \leq (i+1)p(|d|) \leq |d|p(|d|)$. Similarly, it is not hard to find polynomial bounds on the lengths of the proofs of the needed equivalences, and there are only polynomially many of them.

This completes the proof of Theorem 2.2. □

We have handled the case where there are at least two elements in the universe. On the other hand, on the assumption that there is only one element of the universe, we are reduced to propositional logic.

Proposition 2.3 *$\{\forall x, y \; (x = y)\}$ has efficient elimination of definitions if and only if the corresponding assertion holds for propositional logic.*

Proof. Assuming $\forall x, y \; (x = y)$, every atomic formula $R(t_1, \ldots, t_k)$ is equivalent to $R(c, \ldots, c)$, where c is the only element of the universe; $t_1 = t_2$ is always true; and quantifiers have no effect. To be more precise, let "the propositional simplification of ψ" denote the result of deleting all the quantifiers in ψ, replacing all atomic formulae $R(t_1, \ldots, t_k)$ by a propositional variable R, and replacing $t_1 = t_2$ by a fixed tautology. Then any first-order proof of $\forall x, y \; (x = y) \rightarrow \psi$ can be translated efficiently to a propositional proof of the propositional simplification of ψ, and vice-versa. □

This implies that the general problem of eliminating definitions from proofs in pure first-order logic is as hard (and as easy) as the propositional case.

Theorem 2.4 *$\emptyset$ has an efficient elimination of definitions if and only if the corresponding assertion holds for propositional logic.*

Proof. It is a straightforward exercise to check that $\{\varphi \vee \psi\}$ has an efficient elimination of definitions if and only if $\{\varphi\}$ and $\{\psi\}$ both do. In particular, $\emptyset$ has an efficient elimination of definitions if and only if $\{\forall x, y \; (x = y)\}$ and $\{\exists x, y \; (x \neq y)\}$ do. □

As a corollary of Theorem 2.2, we have that one can eliminate $\leftrightarrow$ from standard proof systems with at most a polynomial increase in proof length. For propositional proof systems the proof (due to Reckhow, using a method by Spira; see [9]) is considerably more difficult.

Corollary 2.5 *With any of the standard proof systems for first-order logic with equality given in [15], one can eliminate the propositional connective $\leftrightarrow$ with at most a polynomial increase in proof length.*

Proof. By Theorem 2.2, it suffices to show that one can eliminate $\leftrightarrow$ efficiently in the corresponding proof systems with definitions. Use definitions to translate formulae in the language with $\leftrightarrow$ to the language without: translate $\varphi(\vec{w}) \leftrightarrow \psi(\vec{z})$ to $(R_\varphi(\vec{w}) \rightarrow R_\psi(\vec{z})) \wedge (R_\psi(\vec{z}) \rightarrow R_\varphi(\vec{w}))$, where R_0 and R_1 are defined to be equivalent to the translations of φ and ψ, respectively. By induction one can show that each axiom and rule of inference can then be simulated, with polynomial bounds on the lengths. □

3 Eliminating Skolem functions

The following is the analogue of Definition 2.1 for Skolem functions.

Definition 3.1 *Let Γ be a set of first-order sentences in a language L. Say that Γ has an* efficient elimination of Skolem functions *if there is a polynomial $p(x)$ such that the following holds: whenever $f_0(\vec{x}_0), \ldots, f_k(\vec{x}_k)$ are new function symbols of various arities, $\varphi_0(\vec{x}_0, y), \ldots, \varphi_k(\vec{x}_k, y)$ are formulae such that each φ_i is in the language $L \cup \{f_0, \ldots, f_{i-1}\}$, and d is a proof of a formula ψ in L from*

$$\Gamma \cup \{\forall \vec{x}_0, y\ (\varphi_0(\vec{x}_0, y) \to \varphi(\vec{x}_0, f_0(\vec{x}_0))), \ldots,$$
$$\forall \vec{x}_k, y\ (\varphi_k(\vec{x}_k, y) \to \varphi(\vec{x}_k, f_k(\vec{x}_k)))\},$$

then there is a proof d' of ψ from Γ using only formulae in L, with $|d'| \leq p(|d|)$.

Right off the bat, we have the following.

Proposition 3.2 *$\{\forall x, y\ (x = y)\}$ has an efficient elimination of Skolem functions.*

Proof. Roughly speaking, if c is the only element of the universe, every term can be replaced by c. □

By way of motivation, note that is not hard to show that, say, Zermelo-Fraenkel set theory has an efficient elimination of Skolem functions. Argue as follows. Suppose d is a proof of a formula ψ from the axioms of ZF and some Skolem functions. Let k be a bound on the complexity of the formulae occurring in this proof. In ZF, one can prove that the set of true sentences of complexity at most $k + 1$ is consistent, and hence has a countable model. This countable model has Skolem functions, which can then be used to interpret the proof d.

This example suggests that one way to proceed is to try to determine how little one can get away with in carrying out an internal semantic argument of this kind. The answer turns out to be: very little.

Definition 3.3 *Say a set of sentences Γ* codes finite functions (efficiently) *if for each n there are*

- *a definable element, "$\emptyset_n$";*
- *a definable relation, "$x_0, \ldots, x_{n-1} \in dom_n(p)$";*
- *a definable function, "$eval_n(p, x_0, \ldots, x_{n-1})$"; and*
- *a definable function, "$p \oplus_n (x_0, \ldots, x_{n-1} \mapsto y)$"*

such that, for each n, Γ proves

- $\vec{x} \notin dom(\emptyset_n)$
- $\vec{w} \in dom(p \oplus (\vec{x} \mapsto y)) \leftrightarrow (\vec{w} \in dom(p) \vee \vec{w} = \vec{x})$
- $eval_n(p \oplus_n (\vec{x} \mapsto y), \vec{x}) = y$
- $\vec{w} \neq \vec{x} \to eval_n(p \oplus_n (\vec{x} \mapsto y), \vec{w}) = eval_n(p, \vec{w})$,

and such that the lengths of all the definitions and proofs are bounded by a polynomial in n.

Of course, the intuition is that elements of the universe are assumed to code finite partial functions p; $\emptyset_n$ is the function that is nowhere defined; $eval_n(p, \vec{x})$ returns the value of p at $\vec{x}$; $p \oplus_n (\vec{x} \mapsto y)$ is the modification of p which maps $\vec{x}$ to y; and so on. One could, more generally, assume that the codes are elements of a definable set; but then nothing is lost by taking the other elements of the universe to code the empty function. If one wants polynomial-time translations (and not just bounds on the lengths of proofs) one needs to add the constraint that the definitions and proofs above are polynomial-time computable in n.

These requirements are not strong ones. For example, any sequential theory of arithmetic (in the terminology of [7, 9, 11]) codes finite functions, since one can take such functions to be sequences of tuples $\langle \vec{x}, y \rangle$. Below I will drop the subscripts n in $\emptyset_n$, dom_n, etc. and I will write $p(\vec{x})$ instead of $eval(p, \vec{x})$. Clearly it does not hurt to assume that all these are actually given by symbols in the language.

Theorem 3.4 *Suppose Γ codes finite functions. Then Γ has an efficient elimination of Skolem functions.*

Proof. The proof will occupy most of the remainder of this section. By Proposition 3.2 we can assume that there are at least two elements in the universe, and so, by Theorem 2.2, we can use definitions freely. By way of exposition, I will first focus on the case where $k = 0$, i.e. there is only one Skolem function to eliminate. (This part does not require definitions.) Then I will discuss the steps necessary to eliminate multiple, possibly nested instances Skolem functions. (This is the part that requires definitions.)

Suppose we want to eliminate the use of a single Skolem function, with defining axiom $\forall \vec{x}, y\ (\varphi(\vec{x}, y) \to \varphi(\vec{x}, f(x)))$. Let L_f denote the language $L \cup \{f\}$. I will define a forcing relation in L, for formulae in L_f. I will then show that Γ proves that the Skolem axiom is forced; and that anything in the original language is forced if and only if it is true. Given a proof d of ψ from Γ together with the Skolem axiom, then, Γ proves that ψ is forced, and hence true.

Now for the details. Let the formula $Cond(p)$ in the language L assert that p is a finite approximation to a Skolem function for φ, that is,

$$\forall \vec{x} \in dom(p)\ \forall y\ (\varphi(\vec{x}, y) \to \varphi(\vec{x}, f(x))).$$

Let t be a term in L_f, and let p be a variable not occurring in t. Inductively we will define a term t^p in the language of L, whose free variables are those of t together with p. Intuitively, t^p is the value of t, when f is interpreted by p. At the same time, we will define a relation "t^p is defined," asserting that the value of t^p makes sense. Let

- $x^p \equiv x$, for each variable x (other than p),
- $(g(t_0, \ldots, t_m))^p \equiv g(t_0^p, \ldots, t_m^p)$, for each function symbol g of L, and
- $(f(t_0, \ldots, t_n))^p \equiv p(t_0^p, \ldots, t_n^p)$.

Define "t^p is defined" inductively as follows:

- "x^p is defined" is always true.
- "$(g(t_0, \ldots, t_m))^p$ is defined," where g is a function symbol of L, is true if and only if $t_0^p, \ldots, t_m^p$ are all defined.
- "$(f(t_0, \ldots, t_n))^p$ is defined" is true if and only if $t_0^p, \ldots, t_n^p$ are all defined and $t_0^p, \ldots, t_n^p \in dom(p)$.

If p and q are conditions, say $p \preceq q$, "p is stronger than or equal to q", if p extends q as a function:

$$\forall \vec{x}\ (\vec{x} \in dom(q) \rightarrow \vec{x} \in dom(p) \wedge p(\vec{x}) = q(\vec{x})).$$

Now we can define the relation $p \Vdash \theta$ inductively. We can assume that the language has connectives $\wedge$, $\rightarrow$, $\forall$, and $\neg$, with $\exists$ and $\vee$ defined from these in the usual way.

1. $p \Vdash R(t_0, \ldots, t_m)$ if and only if $\forall q \preceq p\ \exists r \preceq q$ ($t_0^r, \ldots, t_m^r$ are all defined and $R(t_0^r, \ldots, t_m^r)$).
2. $p \Vdash \theta \wedge \eta$ if and only if $p \Vdash \theta$ and $p \Vdash \eta$.
3. $p \Vdash \theta \rightarrow \eta$ if and only if $\forall q \preceq p\ (q \Vdash \theta \rightarrow q \Vdash \eta)$.
4. $p \Vdash \neg\theta$ if and only if $\forall q \preceq p\ q \nVdash \theta$.
5. $p \Vdash \forall x\ \theta$ if and only if $\forall x\ p \Vdash \theta$.

The quantifiers involving q and r above are intended to range over conditions; so, for example, $\forall q \preceq p \ \ldots$ abbreviates $\forall q\ (Cond(q) \wedge q \preceq p \rightarrow \ldots)$. For each θ, the relation $p \Vdash \theta$ is a formula in the language of L whose free variables are those of θ together with p. Note that the length of $p \Vdash \theta$ can be bounded by a polynomial in $|\theta|$ (as well as in $|\varphi|$, which is being held fixed for the moment).

The phrase "θ is forced" and the notation $\Vdash \theta$ abbreviate $\forall p\ (Cond(p) \rightarrow p \Vdash \theta)$. In the lemmata that follow, $p, q, r \ldots$ are assumed to range over conditions. Most of the proofs are routine and standard, modulo the additional notes provided below. It is important to recognize that the lengths of all the proofs alluded to in the statement of the lemmata can be bounded by a polynomial in the length of the assertion being proved, but having stated this up front, I will not bother to repeat it each time.

Lemma 3.5 (monotonicity) *For each formula θ of L_f, Γ proves*

$$p \Vdash \theta \wedge q \preceq p \rightarrow q \Vdash \theta.$$

Lemma 3.6 *For each formula θ of L_f, Γ proves*

$$p \Vdash \theta \leftrightarrow \forall q \preceq p\ \exists r \preceq q\ r \Vdash \theta.$$

Corollary 3.7 *For each formula θ of L_f, Γ proves*

$$p \Vdash (\theta \leftrightarrow \neg\neg\theta).$$

Lemma 3.8 *For any term t of L_f, Γ proves*

$$\forall q\ \exists r \preceq q\ (t^r \textit{ is defined}).$$

Proof. Use induction on the term t. The only interesting case is where t is of the form $f(s_0, \ldots, s_k)$. By the induction hypothesis, we can find an $r' \preceq q$ such that $s_0^{r'}, \ldots, s_k^{r'}$ are all defined. If $s_0^{r'}, \ldots, s_k^{r'} \in dom(r')$, take $r = r'$. Otherwise, if $\exists y\ \varphi(s_0^{r'}, \ldots, s_k^{r'}, y)$, let $r = r' \oplus (s_0^{r'}, \ldots, s_k^{r'} \mapsto y)$, for any such y; and if $\forall y\ \neg\varphi(s_0^{r'}, \ldots, s_k^{r'}, y)$, let $r = r' \oplus (s_0^{r'}, \ldots, s_k^{r'} \mapsto y)$, for any y at all. □

The next two lemmata are proved by induction on s and θ, respectively.

Lemma 3.9 *If t and $s(x)$ are any terms of L_f, Γ proves*

$$t^p = z \rightarrow (s(t)^p = s(z)^p)$$

Lemma 3.10 *If $\theta(x)$ is any formula of L_f and t is any term of L_f then Γ proves*

$$(t^p \textit{ is defined} \wedge t^p = z) \rightarrow (p \Vdash \theta(t) \leftrightarrow p \Vdash \theta(z)).$$

Lemma 3.11 *For each formula θ of L_f, if θ is provable in classical first-order logic, then Γ proves $\Vdash \theta$.*

Proof. The proof is for the most part standard and routine, though one has to be a little bit careful with the quantifier axioms and rules since terms might not always be "defined." To show $\forall x\ \theta(x) \rightarrow \theta(t)$ is forced, let us argue in first-order logic from assumptions in Γ. Suppose $p \Vdash \forall x\ \theta(x)$. By Lemma 3.6 it suffices to show $\forall q \preceq p\ \exists r \preceq q\ \theta(t)$. So suppose $q \preceq p$, and by Lemma 3.8 let $r \preceq q$ be such that t^r is defined. Let $z = t^r$. By monotonicity, $r \Vdash \forall x\ \theta(x)$, so $r \Vdash \theta(z)$. By Lemma 3.10, $r \Vdash \theta(t)$. □

A formula in the original language is forced if and only if it is true.

Lemma 3.12 *For each formula θ of L, Γ proves $(p \Vdash \theta) \leftrightarrow \theta$.*

Proof. Induction on θ. □

The next lemma is the important one: it asserts that the Skolem axiom is forced.

Lemma 3.13 *Γ proves $\Vdash \forall \vec{x}, y\ (\varphi(\vec{x}, y) \to \varphi(\vec{x}, f(\vec{x})))$.*

Proof. Once again, argue in first-order logic, from Γ. Suppose for some $\vec{x}, y$ we have $p \Vdash \varphi(\vec{x}, y)$. By Lemma 3.12, $\varphi(\vec{x}, y)$. By Lemma 3.6, it suffices to show $\forall q \preceq p\ \exists r \preceq q\ q \Vdash \varphi(\vec{x}, f(\vec{x}))$, so suppose $q \preceq p$. If $\vec{x} \in dom(q)$, the fact that q is a condition guarantees $\varphi(\vec{x}, f(\vec{x}))$, and we can take $r = q$; otherwise, take $r = q \oplus (\vec{x} \mapsto y)$. Either way, as above, we have $r \Vdash \varphi(\vec{x}, f(\vec{x}))$, as required. □

Proof of Theorem 3.4, for a single Skolem function f. Suppose there is a proof d of a formula ψ in the language L from finitely many sentences in $\Gamma \cup \{\forall \vec{x}, y\ \ (\varphi(\vec{x}, y) \to \varphi(\vec{x}, f(x)))\}$. By Lemma 3.11, Γ proves that this implication is forced. By Lemmata 3.12 and 3.13, Γ proves that all the hypotheses are forced, so Γ proves that ψ is forced as well. By Lemma 3.12, Γ proves ψ.

Since each the length of each component of the derivation just described can be bounded by a polynomial in $|d|$, so can the entire proof. □

To extend the proof to arbitrary nested definitions of Skolem functions, we need to iterate the forcing definition. A similar iteration was used in [2]; the situation here is easier, since we only have to deal with finite iterations.

Let $d, f_0, \ldots, f_k, \varphi_0, \ldots, \varphi_k$ be as in Definition 3.1. For each $i \leq k$, we will define the notion of an i-condition, an ordering $\preceq_i$ on i-conditions, and a forcing relation $\Vdash_i$ between i-conditions and formulae θ in the language $L \cup \{f_0, \ldots, f_i\}$. An i-condition consists of a sequence $p_0, \ldots, p_i$ of finite functions, with arities corresponding to those of $f_0, \ldots, f_i$. As expected, $p_0, \ldots, p_i \preceq_i q_0, \ldots, q_i$ means that each p_j extends q_j, as above.

The notions $Cond_i$ and $\Vdash_i$ are defined simultaneously, by recursion on i. $Cond_0(p)$ and $p \Vdash_0 \theta$ are defined as above, in the case where there is only one Skolem function. Assuming $Cond_i$ and $\Vdash_i$ have been defined, the relation $Cond_{i+1}(p_0, \ldots, p_{i+1})$ is defined by

$$Cond_i(p_0, \ldots, p_i) \wedge p_0, \ldots, p_i \Vdash_i \forall \vec{x}_{i+1}, y\ (\vec{x}_{i+1} \in dom(p_{i+1}) \wedge \varphi(\vec{x}_{i+1}, y) \to \varphi(\vec{x}_{i+1}, p(\vec{x}))).$$

In the atomic case, assuming $t_0, \ldots, t_m$ are terms in the language of $L \cup \{f_0, \ldots, f_{i+1}\}$, the relation $p_0, \ldots, p_{i+1} \Vdash_{i+1} A(t_0, \ldots, t_m)$ is defined by

$$\forall \vec{q} \preceq \vec{p}\ \exists \vec{r} \preceq \vec{q}\ (t_0^{\vec{r}}, \ldots, t_m^{\vec{r}} \text{ are defined and } A(t_0^{\vec{r}}, \ldots, t_m^{\vec{r}})).$$

The forcing relation is then extended to arbitrary formulae in the language as above. Notice that the relation $\Vdash_i$ is used in the definition of $Cond_{i+1}$, which is in turn used to define $\Vdash_{i+1}$. By introducing new relation symbols to represent the definitions of $Cond_0, \ldots, Cond_k$, we can bound the lengths of all the formulae involved by a polynomial.

Lemma 3.14 *For each $i \leq k$, Lemmata 3.5–3.11 hold for i-conditions, $\preceq_i$, and $\Vdash_i$.*

Lemma 3.15 *For each $i \leq k$, if θ is in the language $L \cup \{f_0, \ldots, f_i\}$, then Γ proves the following:*

$$p_0, \ldots, p_k \Vdash_k \theta \leftrightarrow p_0, \ldots, p_i \Vdash_i \theta.$$

Lemma 3.16 *For each $i \leq k$, Γ proves that the ith Skolem axiom is k-forced.*

Once again, the lengths of the relevant proofs can be bounded by a polynomial in $|d|$. The proof of Theorem 3.4 now follows exactly as in the case of a single Skolem function. □

If a and b are distinct and f is a Skolem function for $(\varphi(\vec{x}) \wedge y = a) \vee (\neg\varphi(\vec{x}) \wedge y = b)$, then $f(\vec{x}) = a$ serves as a definition for $\varphi(\vec{x})$. As a corollary to Theorem 3.4 we have the following:

Corollary 3.17 *Suppose Γ codes finite functions and proves $\exists x, y\ (x \neq y)$. Then one can eliminate arbitrary nested instances of definitions and Skolem functions from proofs in Γ, with a polynomial bound on the increase in the lengths of proofs.*

4 Questions

In standard terminology (e.g. [9, 11]), Section 2 shows that one can eliminate definitions from proofs in first-order logic in polynomial time if and only if extended Frege systems for propositional logic can be p-simulated by Frege systems. Of course, whether or not this is the case is still a major open question. Section 2 shows that Theorem 2.2 and Corollary 2.5 hold for first-order logic with equality. What can one say in the absence of equality?

It is also still open as to whether one can efficiently eliminate even a single Skolem function from proofs in pure logic, or from theories that do not code finite functions.

The elimination of definitions in Section 2 used the law of the excluded middle. As a result, it is open as to whether one has an efficient elimination of definitions in intuitionistic first-order logic. (See also [12] for a discussion of choice functions in the intuitionistic setting.)

This work has been partially supported by NSF grant DMS 0070600. I am grateful to Samuel Buss for advice and suggestions.

References

[1] M. Ajtai. The complexity of the pigeonhole principle. *Proceedings of the IEEEE 29th Annual Symposium on Foundations of Computer Science*, pages 346–355, 1988.

[2] J. Avigad. Formalizing forcing arguments in subsystems of second-order arithmetic. *Annals of Pure and Applied Logic*, 82:165–191, 1996.

[3] J. Avigad. Interpreting classical theories in constructive ones. *Journal of Symbolic Logic*, 65:1785–1812, 2000.

[4] J. Avigad. Algebraic proofs of the cut-elimination theorems. To appear in the *Journal of Logic and Algebraic Programming*.

[5] J. Avigad. Weak theories of nonstandard arithmetic and analysis. To appear in S. Simpson, editor, *Reverse Mathematics 2001*.

[6] P. Clote and J. Krajíček. *Arithmetic, Proof Theory, and Computational Complexity*. Oxford University Press, Oxford, 1993.

[7] P. Hájek and P. Pudlák. *Metamathematics of first-order arithmetic*. Springer, Berlin, 1993.

[8] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 2. Springer, Berlin, second edition, 1970.

[9] J. Krajíček. *Bounded Arithmetic, Propositional Logic, and Complexity Theory*. Cambridge University Press, Cambridge, 1995.

[10] J. Paris and A. Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic*, Lecture Notes in Mathematics, v. 1130, pages 317–340. Springer, Berlin, 1985.

[11] P. Pudlák. The lengths of proofs. In S. Buss, editor, *The Handbook of Proof Theory*, pages 547–637. North-Holland, Amsterdam, 1998.

[12] H. Schwichtenberg. Logic and the axiom of choice. In M. Boffa et al, editor, *Logic Colloquium '78*, pages 351–356. North-Holland, Amsterdam, 1979.

[13] J. R. Shoenfield. *Mathematical Logic*. Addison Wesley, Reading, Massachussets, 1967.

[14] G. Takeuti. *Proof Theory*. North-Holland, Amsterdam, second edition, 1987.

[15] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, Cambridge, 1996.

On the decision problem for the guarded fragment with transitivity *

Wiesław Szwast and Lidia Tendera
Institute of Mathematics
Opole University, Oleska 48, 45-052 Opole, Poland
{szwast,tendera}@cs.uni.opole.pl

Abstract

The guarded fragment with transitive guards, [GF+TG], *is an extension of* GF *in which certain relations are required to be transitive, transitive predicate letters appear only in guards of the quantifiers and the equality symbol may appear everywhere. We prove that the decision problem for* [GF+TG] *is decidable. This answers the question posed in* [11]. *Moreover, we show that the problem is* 2EXPTIME-*complete. This result is optimal since satisfiability problem for* GF *is* 2EXPTIME-*complete* [12]. *We also show that the satisfiability problem for two-variable* [GF+TG] *is* NEXPTIME-*hard in contrast to* GF *with bounded number of variables for which the satisfiability problem is* EXPTIME-*complete.*

1 Introduction

Modal logic, that in medieval times was studied by philosophers, in the last decades became a subject of interest for computer scientists. Modal logic has applications in many areas of computer science including artificial intelligence [5, 21], program verification [8, 24, 23], database theory [7, 20] and distributed computing [6, 16].

Propositional modal logic possesses useful model-theoretic and good algorithmic properties, like finite axiomability, Craig interpolation, Beth definability and decidability for validity. The tractability of modal logic was partially explained when D. Gabbay [10] showed that modal logic can be embedded in FO^2, the fragment of first order logic with two variables, that is decidable. The decidability of FO^2 was studied by D. Scott [25] who proved that the satisfiability problem for FO^2 without equality is decidable, by M. Mortimer [22] who proved that FO^2 with equality has a finite model property, and by E. Grädel, Ph. Kolaitis and M. Vardi [13] who proved the exponential model property for FO^2. The last result together with the result by H. Lewis [19] implies that the satisfiability problem for FO^2 is NEXPTIME-complete.

FO^2 can be used as a representative language also for a number of knowledge representation logics (description logics) [2]. Moreover, many extensions of modal logics that are not fragments of FO^2 can easily be embedded in some extensions of FO^2, for example, CTL and the μ-calculus can be treated as FO^2 with a fixed-point operator [28] and many powerful variants of the description logics can be embedded in C^2, the extension of FO^2 with counting quantifiers, or in C^2 with transitivity [4].

Although the translation of modal logic to FO^2 explains some good properties of modal logic, it does not work in the same way for distinct extensions of modal logic. In particular, CTL has an EXPTIME-complete validity problem but FO^2 with a fixed-point operator was shown to be undecidable [14]. Similarly, Immerman and Vardi proved [17] that CTL can be embedded in FO^2 with a transitive closure operator that is again undecidable. In addition, FO^2 has a very poor proof theory so it cannot be seen as a natural fragment of predicate logic extending modal logic and capturing all nice properties of modal logic. The model theoretic reason for the nice behavior of modal logic was recently given in [28] where Vardi answers the explicitly asked question 'Why is modal logic so robustly decidable?'

The guarded fragment. In 1996, H. Andréka, J. van Benthem and I. Németi [1] introduced the *guarded fragment* of first-order logic, GF, in order to explain and generalize the good properties of modal logic. GF consists of first-order formulas where all quantifiers are appropriately relativized by atoms but neither the pattern of alternations of quantifiers nor the number of variables is restricted. Andréka et al. showed that modal logic can be embedded in GF and they argued convincingly that GF inherits the nice properties of

*This research was supported by KBN grant 2 P03A 018 18

0-7695-1281-X/01 $10.00 © 2001 IEEE

modal logic. The nice behavior of GF was confirmed by Grädel [12] who proved that the satisfiability problem for GF is complete for double exponential time and complete for exponential time, when the number of variables is bounded.

In order to express certain properties of temporal logic, GF was later generalized by van Benthem [27] to the *loosely guarded fragment*, LGF, where all quantifiers are relativized by conjunctions of atoms. Most of the properties of GF generalize to LGF.

In [28] Vardi argued that one of the main reasons for the nice behavior of modal logics is the tree model property. It was proved [12] that both GF and LGF also have a tree-model property analogous to the tree-model property for modal logic; in addition, GF has the finite model property. So, having proved several good properties of the guarded fragment one could expect that the same will hold for some extensions of GF, similarily to the basic modal logic that remains decidable and of fairly low complexity under the addition of a variety of operators and features, such as counting modalities, transitive closure modalities and conditions on the accessibility relation. In [15] Grädel and Walukiewicz proved that extending GF with fixed point operators one gets still a decidable logic. Moreover, they proved that the satisfiability problem for GF with fixed points can be decided in the same time as for pure GF. The same is true for GF with bounded number of variables.

The transitivity constraints. The extension of the guarded fragment by transitivity seems to be a natural representative language e.g. for multi-modal logics of type K4, S4 or S5. These multi-modal logics are used to formalize epistemic logics [9]. Unfortunately, Grädel [12] proved that

- GF^3, the three-variable fragment of GF, with transitive relations (or with counting quantifiers) is undecidable.

The three-variable guarded fragment may be too strong to represent modal logics, since, as it is mentioned at the begining, two variables suffice. However, in [11], besides other results, H. Ganzinger, C. Meyer and M. Veanes improved the result by Grädel [12] showing that even

- GF^2 with transitive relations and without equality is undecidable.

In [11] Ganzinger et al. studied decidability issues for the extension of GF with transitivity constraints and they proposed a logic that is an extension of GF in which transitive predicate letters appear only in guards of the quantifiers whereas non-transitive predicates and the equality symbol may appear everywhere. In this paper we denote it by [GF+TG] and we call it the *guarded fragment with transitive guards*. [GF+TG] is powerfull enough to be used as a representative language for multi-modal logics of type K4, S4 or S5, since when encoding them in the first order logic the predicate letters corresponding to accessibility relations occur only in guards. By [GF^2+TG] we denote the two-variable fragment of [GF+TG] and by *monadic*-[GF^2+TG] - the fragment of [GF^2+TG] in which all non-unary predicate letters may appear in guards only. Ganzinger et al. [11]gave a nice proof of theorem that

- monadic-[GF^2+TG] is decidable.

and they asked the following two questions:

1. What is the complexity of monadic-[GF^2+TG]? (The proof in [11] proceeds through a reduction to the monadic theory of a tree, SkS, and hence no special complexity bound has been given there.)
2. Is satisfiability of the full [GF+TG] decidable?

This paper. We prove that the satisfiability of [GF+TG] can be decided in deterministic double exponential time. Since [GF+TG] is an extension of GF we immediately get that [GF+TG] is 2EXPTIME-complete. So, similarly to GF with fixed point operators, we do not have to pay more for adding transitive guards and this makes [GF+TG] the right counterpart of certain extensions of modal logics.

We also prove that the satisfiability problem for monadic-[GF^2+TG] with equality is hard for nondeterministic exponential time. This is proved by a reduction of FO^2-sentences to [GF^2+TG]-sentences that preserves satisfiability. This reduction is based on an observation that in monadic-[GF^2+TG] we are able to define cliques that are big enough to enclose models for FO^2-sentences. Then NEXPTIME-hardness of [GF^2+TG] follows from NEXPTIME-hardness of FO^2. This result has been recently improved by E. Kieroński [18] who showed that monadic-[GF^2+TG] even without equality, is hard for EXPSPACE. These results are rather surprising since both GF and GF with fixed point operators when restricted to bounded number of variables are EXPTIME-complete and as we show in the main part of the paper the complexity for the full [GF+TG] is exactly the same as for GF.

It is worth noticing that [GF+TG] and [GF^2+TG] are strictly more expressive than the monadic subclass. As an example of a [GF^2+TG]-sentence that cannot be expressed in monadic-[GF^2+TG] one can

take the sentence defining cliques since, as we observed in [26], every satisfiable monadic-[GF^2+TG] sentence has a model without symmetric edges.

The proof of the decidability is very technical. To obtain the decision procedures we apply new techniques inspired by the standard methods of modal logic that were used for establishing positive results for GF and its extensions, namely the tree-model property and bisimulation.

As the first step we observe that the size of cliques of elements connected with transitive relations in models of [GF+TG]-sentences can be bounded. Using this observation, we define *ramified* structures that have cliques of exponential size (with respect to the signature), and that have only disjoint transitive paths for distinct transitive predicate letters. Then, for a fixed element a of a ramified structure, we define a *flower* that contains information about the cliques of the element a and the colors of elements connected with a by non-symmetric edges.

As the next step we observe that the set of flowers realized in a ramified model for a [GF+TG]- sentence satisfies some properties, for example, if two distinct elements are connected with a non-symmetric, transitive predicate, then every color connected with the first element has to be connected with the second one. We collect several such properties in the definition of a special set of flowers named a *carpet* and we show that these properties are necessary and sufficient for existence of a model for a [GF+TG]-sentence. In the proofs we do not construct models that explicitly possess the tree-model property but the models are "tree-controlled:" during the construction every element is added as a child of an element on a fixed level of a tree. The proof that a (ramified) structure is a model for a [GF+TG]-sentence can be seen as an application of bisimulation but where at every moment we need to care about a big set of cliques of elements that lay on one transitive path.

The final step is based on the facts that the size of a flower is exponential and the number of all flowers is double exponential, and this allows us to build an alternating test for satisfiability for [GF+TG]-sentences that uses exponential space.

2 Preliminaries

By FO^k we denote the class of first order sentences with k variables over a relational signature. The *guarded fragment*, GF, of first-order logic with no function symbols of arity greater than 0, is defined as the least set of formulas such that

(1) every atomic formula belongs to GF,

(2) GF is closed under logical connectives $\neg, \vee, \wedge, \rightarrow$,

(3) if $\mathbf{x}, \mathbf{y}$ are tuples of variables, $\alpha(\mathbf{x}, \mathbf{y})$ is atomic and $\psi(\mathbf{x}, \mathbf{y})$ is a formula of GF with free variables contained in $\{\mathbf{x}, \mathbf{y}\}$, then the formulas

$$\forall \mathbf{y} \alpha(\mathbf{x}, \mathbf{y}) \rightarrow \psi(\mathbf{x}, \mathbf{y}),$$

$$\exists \mathbf{y} \alpha(\mathbf{x}, \mathbf{y}) \wedge \psi(\mathbf{x}, \mathbf{y})$$

belong to GF.

The atom $\alpha(\mathbf{x}, \mathbf{y})$ in the above formulas is called the *guard* of the quantifier.

In this paper we admit conditions stating that some binary predicate T is transitive, we express these conditions by *"T is transitive"* and we let $Trans[T_1, \ldots, T_m]$ stand for the condition that each T_i is transitive. In this case we also say that T is a *transitive predicate letter*. Denote by [GF+TG] the set of sentences contained in GF with all transitive predicate letters appearing in guards only and where the equality symbol can appear everywhere and let [GF^k+TG]=FO$^k \cap$[GF+TG].

Let σ be a relational signature. If $\mathbf{x}$ is a sequence of variables $(x_1, \ldots, x_k)$, then a *k-type* $t(\mathbf{x})$ is a maximal consistent set of atomic and negated atomic formulas over σ in the variables of $\mathbf{x}$. A type t is often identified with the conjunction of formulas in t. If not stated otherwise, 1-types are types of the variable x and 2-types are types of the variables x and y.

Let $\psi(\mathbf{x})$ be a quantifier-free formula in the variables of $\mathbf{x}$. We say that a type t *satisfies* ψ if ψ is true under the truth assignment that assigns true to an atomic formula precisely when it is a member of t and this is denoted by $t \models \psi$.

A k-type s is a *reduction* of an m-type t, if there exists a substitution $\rho : \{1, \ldots, k\} \mapsto \{1, \ldots, m\}$ such that $t(x_1, \ldots, x_m) \models s(x_{\rho(1)}, \ldots, x_{\rho(k)})$. A $k+1$-type t *extends* a k-type s if $s \subseteq t$ and a $k+1$-type t *properly extends* a k-type s if t extends s and for every $i \leq k$, t contains the formula $x_i \neq x_{k+1}$.

If $\mathfrak{A}$ is a σ-structure with the universe A, and if $\mathbf{a} \in A^k$, then we denote by $tp^{\mathfrak{A}}(\mathbf{a})$ the unique k-type realized by $\mathbf{a}$ in $\mathfrak{A}$. If $B \subseteq A$ then $\mathfrak{A}{\restriction}B$ denotes the substructure of $\mathfrak{A}$ restricted to the universe B.

If $\mathfrak{A}$ and $\mathfrak{B}$ are σ-structures, $a \in A$ and $b \in B$ then we write $(\mathfrak{A}, a) \cong (\mathfrak{B}, b)$ to say that there is an isomorphism f of the structures $\mathfrak{A}$ and $\mathfrak{B}$ such that $f(a) = b$.

3 The normal form

In [12] Grädel showed a reduction that transforms each GF-sentence to a sentence in *normal form* that

preserves satisfiability. In this section we show a similar reduction that additionally keeps the number of variables and the arity of predicate letters of the input sentence.

Definition 3.1 *A* [GF+TG]*-sentence* Δ *is in* normal form *iff it is a conjunction of sentences of the following form:*

(n1) $\exists x(\alpha(x) \wedge \psi(x))$,

(n2) $\forall \mathbf{x}(\alpha(\mathbf{x}) \rightarrow \exists \mathbf{y}(\beta(\mathbf{x},\mathbf{y}) \wedge \psi(\mathbf{x},\mathbf{y})))$,

(n3) $\forall \mathbf{x}(\alpha(\mathbf{x}) \rightarrow \psi(\mathbf{x}))$,

where $\mathbf{y} \cap \mathbf{x} = \emptyset$, α *and* β *are atomic formulas,* ψ *is quantifier-free and it contains no transitive predicate letter.*

We have the following lemma.

Lemma 3.2 *With every* [GF^k+TG]*-sentence* Γ *of the length* n *over a signature* τ *one can effectively associate a set* $\mathbf{\Delta}$ *of* [GF^k+TG]*-sentences in normal form over an extended signature* σ, $\mathbf{\Delta} = \{\Delta_1, \ldots, \Delta_d\}$, *such that*

(1) Γ *is satisfiable if and only if* $\bigvee_{i \leq d} \Delta_i$ *is satisfiable,*

(2) $d \leq O(2^n)$ *and for every* $i \leq d$, $|\Delta_i| = O(n \log n)$,

(3) $\mathbf{\Delta}$ *can be computed deterministically in exponential time and every sentence* Δ_i *can be computed in polynomial time with respect to* n.

4 An example

In this section we give an example of a sentence in [GF^2+TG] defining cliques. Hence the class [GF^2+TG] is strictly more expressive than GF and monadic-[GF^2+TG] since, as we observed in [26], every satisfiable monadic-[GF^2+TG]-sentence has a model without symmetric edges.

Let $\sigma(k) = \{T, U_1, \ldots, U_k\}$, where T is a transitive predicate letter and U_i are unary predicate letters and let $\Gamma(k)$ be the conjunction of the following clauses.

(e1) $\forall x \exists y\ (T(x,y) \wedge \bigwedge_{i=1}^{k} U_i(y))$,

(e2) $\forall x \exists y\ (T(y,x) \wedge \bigwedge_{i=1}^{k} U_i(y))$,

(e3) $\forall x,y\ T(x,y) \rightarrow (\bigvee_{i=1}^{k}(U_i(x) \leftrightarrow \neg U_i(y)) \vee x = y)$,

(e4) $Trans[T]$.

Note that $\Gamma(k)$ is a GF-sentence since every sentence of the form: $\forall x \exists \mathbf{y}(\alpha(x,\mathbf{y}) \wedge \psi(x,\mathbf{y}))$ can be written as $\forall x((x = x) \rightarrow \exists \mathbf{y}(\alpha(x,\mathbf{y}) \wedge \psi(x,\mathbf{y})))$. One can check that $\Gamma(k)$ is satisfiable and in every model for $\Gamma(k)$, T is an equivalence relation with equivalence classes of cardinality bounded by 2^k.

In [13] Grädel, Kolaitis and Vardi prove that FO^2 has the exponential model property: there is a constant c such that every satisfiable FO^2-sentence Φ has a model of cardinality at most $2^{c|\Phi|}$.

Let Φ be a FO^2-sentence over a signature τ in Scott's form $\forall x,y\ \phi(x,y) \wedge \bigwedge_i \forall x \exists y\ \phi_i(x,y)$, where $\phi(x,y)$ and $\phi_i(x,y)$ are quantifier-free. Let T be a new binary predicate letter and let Φ' be the following sentence:

$$\forall x,y\ (T(x,y) \rightarrow \phi(x,y)) \wedge \bigwedge_i \forall x \exists y\ (T(x,y) \wedge \phi_i(x,y)).$$

Define the sentence Ψ over the signature $\sigma = \tau \cup \sigma(k)$: $\Psi = \Phi' \wedge \Gamma(k)$, where $k = c \cdot |\Phi|$ and c is given by the exponential model property for FO^2. We have

(1) Ψ is satisfiable if and only if Φ is satisfiable,

(2) $|\Psi| = O(|\Phi| \log(|\Phi|))$ and Ψ is computable in polynomial time with respect to $|\Phi|$,

So, we have proved:

Theorem 4.1 SAT([GF^2+TG]) *is* NEXPTIME*-hard.*

5 The two-variable case

In this section we are concerned with the signature $\sigma = \{U_1, \ldots, U_?, B_1, \ldots, B_?\}$, where U_i is a unary predicate letter, B_i is a binary predicate letter. We do not allow Boolean predicates, function symbols and constants. Assume that $T_1, \ldots, T_m$ are all the transitive predicate letters of σ. Let $M = \{1, \ldots, m\}$.

We reserve the letter T to denote transitive predicates. So, when the predicate letter T or T_i appears in a sentence, then the sentence includes as a conjunct $Trans[T]$ or $Trans[T_i]$, even if this is not written explicitely. Additionally, we allow only 2-types $t(x,y)$ which contain the formula $(x \neq y)$ and we consider structures that have at least two elements.

Remark Although we do not allow predicate letters of arity greater than two, it is possible to transform every two-variable sentence that use these predicate letters to a sentence over a signature containing predicate letters of arity at most two (cf [13]). Moreover, the main part of the new sentence has the same form as the original one and every conjunct that was added during the transformation is a GF^2-sentence with binary predicate letters only. So, with respect to satisfiability, the language σ can be bounded without loss of generality.

In this section we assume that the conjuncts of a [GF^2+TG]-sentence in normal form have the following form (cf. Definition 3.1):

(n1) $\exists x \psi(x)$,

(n2) $\forall x(\alpha(x) \to \exists y(\beta(x,y) \wedge \psi(x,y)))$,

(n3) $\forall x \forall y(\alpha(x,y) \to \psi(x,y))$.

5.1 Ramified models for [GF^2+TG]-sentences

In Section 4 we proved that [GF^2+TG]-sentences can define cliques. In this section we show that the size of cliques of elements connected with transitive relations can be bounded. Using this observation, we define *ramified* structures that have cliques of exponential size (with respect to the signature), and that have only disjoint transitive paths for distinct transitive predicate letters.

Definition 5.1 *(1) A 2-type $t(x,y)$ is* single-transitive *if there exists exactly one transitive predicate letter T such that $t \models T(x,y) \vee T(y,x)$. In this case we also say that t is T-single-transitive. Additionally, if $t \models T(x,y) \wedge T(y,x)$ then t is* symmetric*, otherwise, t is* oriented.

(2) A 2-type $t(x,y)$ is transitive-less *if all the two-variable formulas of $t(x,y)$ containing transitive predicate letters are negated.*

(3) Let $v(x), w(y)$ be 1-types. A negative link of v, w*, denoted by $\overline{v,w}$, is the unique 2-type containing $v(x)$, $w(y)$ and no atomic two-variable formula.*

Definition 5.2 *Let $\mathfrak{A}$ be a σ-structure, B be a binary predicate letter in σ and $\mathfrak{C}$ be a substructure of $\mathfrak{A}$. We say that $\mathfrak{C}$ is a* B-clique *if for every $a, b \in C$ we have $\langle a,b\rangle \in B^{\mathfrak{A}}$.*

Let $a \in A$. We denote by $[a]^{\mathfrak{A}}_B$ the maximal B-clique containing a, provided it exists and B is a transitive predicate letter. In other cases, $[a]^{\mathfrak{A}}_B$ is the one-element structure $\mathfrak{A}\restriction\{a\}$.

Observe that the structure $[a]^{\mathfrak{A}}_B$ need not be a clique even in the case when B is a transitive predicate letter. This happens when there is no element $b \in \mathfrak{A}$ such that $\langle a,b\rangle \in B^{\mathfrak{A}}$.

Definition 5.3 *Let Δ be a [GF^2+TG]-sentence in normal form over σ. A σ-structure $\mathfrak{R}$ is a* ramified model *for Δ if the following conditions hold:*

(1) $\mathfrak{R} \models \Delta$,

(2) for every $a, b \in R$ such that $a \neq b$, $tp^{\mathfrak{R}}(a,b)$ is either a single-transitive or a transitive-less type,

(3) for every $i, j \in M$ such that $i \neq j$, for every $a, b, c \in R$, $b \neq a, c \neq a$, if $b \in [a]^{\mathfrak{R}}_{T_i}$ and $c \in [a]^{\mathfrak{R}}_{T_j}$ then $tp^{\mathfrak{R}}(b,c) = \overline{tp^{\mathfrak{R}}(b), tp^{\mathfrak{R}}(c)}$,

(4) for every $a \in R$, for every $T \in \sigma$, the cardinality of $[a]^{\mathfrak{R}}_T$ is bounded by $2^{O(card(\sigma))}$.

In the above definition we have introduced one of the key notions for this paper. We have the following theorem.

Theorem 5.4 *Every satisfiable* [GF^2+TG]*-sentence Δ in normal form has a ramified model.*

As we will see later, a ramified model is *tree-controlled*, what means, that if we want to build it, we are able to treat the model as a tree, i.e. the universe is partitioned into levels and all the witnesses of elements lying on a given level are their immediate successors.

Before giving the proof we present technical lemmas and introduce some notions that will be useful later.

Definition 5.5 *Let t be a 2-type over σ and B be a binary predicate letter in σ.*

*A B-*slice of t*, denoted by $\overleftrightarrow{t,B}$, is the unique 2-type obtained from t by replacing every atomic formula of the form $T_i(x,y)$ and $T_i(y,x)$, where $T_i \neq B$, by the formula $\neg T_i(x,y)$ and $\neg T_i(y,x)$, respectively.*

Let $\mathcal{T}$ be a set of types. We denote by $\overline{\mathcal{T}} = \{v : v(x) \in \mathcal{T}\} \cup \{\overleftrightarrow{t,B} : t(x,y) \in \mathcal{T}$ and $B \in \sigma\} \cup \{\overline{v,w} : v(x), w(x) \in \mathcal{T}\}$.

Note, that if B is not a transitive predicate letter, then $\overleftrightarrow{t,B}$ is a transitive-less type. On the other hand, when considering $\overleftrightarrow{t,T}$, the only possible appearance of an atomic formula containing a transitive predicate letter is $T(x,y)$ and/or $T(y,x)$, provided the type t contains $T(x,y)$ and/or $T(y,x)$.

Definition 5.6 *Let Δ be a [GF^2+TG]-sentence and let $\mathcal{T}$ be a set of 1-types and 2-types. We say that $\mathcal{T}$ is Δ-acceptable if*

(1) $\mathcal{T}$ is closed under reductions,

(2) for every conjunct of Δ of the form (n1) $\exists x \psi(x)$ *there exists a 1-type $s \in \mathcal{T}$ such that $s \models \psi(x)$,*

(3) for every 1-type $s \in \mathcal{T}$, for every conjunct of Δ of the form (n2) $\forall x(\alpha(x) \to \exists y(\beta(x,y) \wedge \psi(x,y)))$*, there exists $t \in \mathcal{T}$ such that t extends s and $t \models \alpha(x) \to (\beta(x,y) \wedge \psi(x,y))$.*

(4) for every 2-type $t \in \mathcal{T}$, for every conjunct of Δ of the form (n3) $\forall x \forall y(\alpha(x,y) \rightarrow \psi(x,y))$, *we have* $t \models (\alpha(x,y) \rightarrow \psi(x,y)) \wedge (\alpha(x,x) \rightarrow \psi(x,x))$,

The following observation is an easy consequence of the above definitions.

Proposition 5.7 *Let Δ be a* [GF^2+TG]*-sentence in normal form, $\mathfrak{A}$ be a σ-structure and $\mathcal{T}^{\mathfrak{A}}$ be the set of all the 1- and 2-types realized in $\mathfrak{A}$.*

(1) $\mathfrak{A} \models \Delta$ if and only if $\mathcal{T}^{\mathfrak{A}}$ is Δ-acceptable, every element of $\mathfrak{A}$ has a witness for every conjunct of the form (n2) *and every $T \in \sigma$ has a transitively closed interpretation in $\mathfrak{A}$.*

(2) For every set of types $\mathcal{T}$, $\mathcal{T}$ is Δ-acceptable if and only if $\overline{\mathcal{T}}$ is Δ-acceptable.

Definition 5.8 *Let Δ be a* [GF^2+TG]*-sentence in normal form, let $\mathfrak{A}$ be a model for Δ and let $p \in A$.*

*We say that a σ-structure $\mathfrak{D}$ is a T-*petal of $[p]_T^{\mathfrak{A}}$ *if there exists a function G such that $G : D \mapsto [p]_T^{\mathfrak{A}}$ and the following conditions hold:*

(p1) $card(D) = 2^{O(card(\sigma))}$,

(p2) *every 1-type realized in $[p]_T^{\mathfrak{A}}$ is also realized in $\mathfrak{D}$ and the function G preserves 1-types,*

(p3) *there is an element $d \in D$ such that $G(d) = p$,*

(p4) *every 2-type realized in $\mathfrak{D}$ is a T-slice of some type realized in $[p]_T^{\mathfrak{A}}$,*

(p5) *for every $a \in D$, for every conjunct γ of Δ of the form (n2) $\forall x(\alpha(x) \rightarrow \exists y(\beta(x,y) \wedge \psi(x,y)))$, where β contains T, if there exists a witness of $G(a)$ for γ in $[p]_T^{\mathfrak{A}}$ then there exists a witness of a for γ in $\mathfrak{D}$.*

*For a predicate letter B that is not transitive, a structure $\mathfrak{D}$ is a B-*petal of $[p]_B^{\mathfrak{A}}$ *if $\mathfrak{D} \cong [p]_B^{\mathfrak{A}}$.*

Lemma 5.9 *Let Δ be a* [GF^2+TG]*-sentence in normal form, let $\mathfrak{A}$ be a model for Δ and let $p \in A$.*

Then, for every binary predicate letter B, there exists a B-petal of $[p]_B^{\mathfrak{A}}$.

Proof (Sketch) The case when B is not a transitive predicate letter is obvious.

The construction of the required B-petal in case when B is a transitive predicate letter is a subtle modification of the construction given in [13] in the proof that every first-order two-variable sentence has a model of exponential size with respect to the length of the sentence; we give it here for the sake of completeness.

Let Δ be a [GF^2+TG]-sentence in normal form and let T be a transitive predicate letter. To explain the idea of the proof we will use the following notions.

If $\mathfrak{C}$ is a σ-structure, then a *T-local King of $\mathfrak{C}$* is an element of E with the unique 1-type realized in $\mathfrak{C}$, a *T-local Noble of $\mathfrak{C}$* is an element b of E which is necessary for a local King $a \in E$ with respect to a conjunct of the form (n2) $\forall x(\alpha(x) \rightarrow \exists y(\beta(x,y) \wedge \psi(x,y)))$, where $\beta(x,y)$ contains T, and *T-local Plebeians* are the rest of elements of E.

Let $\mathfrak{A}$ be a model for Δ and let $p \in A$. The set

$$D = K \dot{\cup} N \dot{\cup} P_1 \dot{\cup} P_2 \dot{\cup} P_3$$

will be defined as the universe of the required structure $\mathfrak{D}$ - the T-petal of $[p]_T^{\mathfrak{A}}$. The above sets will be the sets of T-local Kings, Nobles and Plebeians of $\mathfrak{D}$; they will play the role of T-local Kings, Nobles and Plebeians of $[p]_T^{\mathfrak{A}}$. Moreover, the set P_1 (P_2 and P_3) consists of elements that are necessary for elements of N (P_1, P_2) with respect to a conjunct of the form (n2) $\forall x(\alpha(x) \rightarrow \exists y(\beta(x,y) \wedge \psi(x,y)))$, where $\beta(x,y)$ contains T. ■

To simplify the presentation of the technical proofs we will use the following special notation.

Definition 5.10 *If $\mathfrak{A}$ and $\mathfrak{B}$ are σ-structures, $a \in A$, $b \in B$ and $tp^{\mathfrak{A}}(a) = tp^{\mathfrak{B}}(b)$, then we denote by $\mathfrak{A}(a) \circ \mathfrak{B}(b)$ the partially defined structure $\mathfrak{R}$ with the universe $A \cup B \setminus \{b\}$ such that for every $c, d \in R$*

- *if $c, d \in A$ then $tp^{\mathfrak{R}}(c,d) = tp^{\mathfrak{A}}(c,d)$,*
- *if $c, d \in B$, $c \neq b$, $d \neq b$, then $tp^{\mathfrak{R}}(c,d) = tp^{\mathfrak{B}}(c,d)$,*
- *if $c \in B$, $c \neq b$, then $tp^{\mathfrak{R}}(a,c) = tp^{\mathfrak{B}}(b,c)$,*
- *if $c \in A$, $d \in B$, $c \neq a$, then $tp^{\mathfrak{R}}(c,d)$ is not defined.*

Now, we are ready to give the proof of Theorem 5.4.

Proof of Theorem 5.4.

Let Δ be a [GF^2+TG]-sentence in normal form, let $\mathfrak{A} \models \Delta$, let $\mathcal{T}^{\mathfrak{A}}$ be the set of 1- and 2-types realized in $\mathfrak{A}$ and let $\mathcal{T} = \overline{\mathcal{T}^{\mathfrak{A}}}$. By Proposition 5.7, the set $\mathcal{T}$ is Δ-acceptable.

We will construct a ramified model of Δ, $\mathfrak{R}$, in which every 2-type is taken from the set $\mathcal{T}$. Every element of R will have a corresponding element in A realizing the same 1-type. The correspondence will be given by a function H, $H : R \mapsto A$, that preserves 1-types and witnesses, i.e. $tp^{\mathfrak{R}}(a) = tp^{\mathfrak{A}}(H(a))$ and if b is a witness of a in $\mathfrak{R}$ for a conjunct of Δ of the form (n2), then $H(b)$ is a witness of $H(a)$ in $\mathfrak{A}$ for the same conjunct.

The structure $\mathfrak{R}$ will be built in stages. In every stage k the structure $\mathfrak{R}_{k-1}$ constructed in stage $k-1$

will be extended to $\mathfrak{R}_k$ by adding new elements to the k-th level of $\mathfrak{R}$, L_k, as witnesses of all the elements of L_{k-1} for the conjuncts of the form (n2). If $a \in L_{k-1}$ and there is no witness $b \in \mathfrak{R}_k$ of a for a conjunct γ of the form (n2), then we add a new element b and put $H(b)$ as a witness of $H(a)$ for γ in $\mathfrak{A}$. The element b is added to L_k together with a structure $\mathfrak{D}$ - a B-petal of the clique $[H(b)]^{\mathfrak{A}}_B$, and the function H is extended for elements of D by the function G given by Definition 5.8. Additionally, to ensure that the same cliques are not added more than once for the same element, when an element a is added to $\mathfrak{R}$ together with its B-clique, then a is *canceled with respect to* B.

So, in every stage k, the following conditions will hold:

(m1) $\mathfrak{R}_k \models \gamma$, for every conjunct γ of the form (n1),

(m2) every 2-type realized in $\mathfrak{R}_k$ is in $\mathcal{T}$,

(m3) for every $a \in L_{k-1}$, for every conjunct γ of the form (n2), there is a witness of a for γ in $\mathfrak{R}_k$,

(m4) for every $T \in \sigma$, the interpretation of T in $\mathfrak{R}_k$ is transitive,

(m5) for every $a, b \in \mathfrak{R}_k$

- $tp^{\mathfrak{R}_k}(a) = tp^{\mathfrak{A}}(H(a))$,
- if b is a witness of a in $\mathfrak{R}_k$ for γ of the form (n2), then $H(b)$ is a witness of $H(a)$ in $\mathfrak{A}$ for γ,
- for every $B \in \sigma$, if $\mathfrak{R}_k \models B(a,b)$ and $a \neq b$ then $tp^{\mathfrak{R}_k}(a,b) = \overleftrightarrow{tp^{\mathfrak{A}}(H(a), H(b)), B}$,

(m6) for every $a \in R_k$, for every $b \in L_k$, for every $T \in \sigma$, if b is not canceled with respect to T, then $tp^{\mathfrak{R}_k}(a,b)$ is either transitive-less or T-single-transitive oriented,

(m7) for every $i, j \in M$ such that $i \neq j$, for every $a, b, c \in R_k$, $b \neq a, c \neq a$, if $b \in [a]^{\mathfrak{R}_k}_{T_i}$ and $c \in [a]^{\mathfrak{R}_k}_{T_j}$ then $tp^{\mathfrak{R}_k}(b,c) = \overline{tp^{\mathfrak{R}_k}(b), tp^{\mathfrak{R}_k}(b)}$,

(m8) for every $a \in R_k$, for every $T \in \sigma$, if a was canceled with respect to T in stage i, $i \leq k$, then $[a]^{\mathfrak{R}_i}_T$ is a T-petal of $[H(a)]^{\mathfrak{A}}_T$ and $[a]^{\mathfrak{R}_i}_T = [a]^{\mathfrak{R}_k}_T$.

Observe that if it is possible to construct a structure $\mathfrak{R}$ that satisfies the conditions (m1) -(m4) then, by part 1 of Proposition 5.7, the structure $\mathfrak{R}$ will be a model for Δ and additionally, by (m2), (m7) and (m8), it will be a ramified model.

The following procedure builds the required structure in a possibly infinite number of stages.

Stage 0. Let $L_0 = R_0 = \emptyset$.

1. For every conjunct $\gamma \in \Delta$ of the form (n1) $\exists x \psi(x)$,
   (a) find $d_\gamma \in A$ such that $\mathfrak{A} \models \psi(d_\gamma)$,
   (b) add a new element b to L_0,
   (c) put $H(b) = d_\gamma$ and put $tp^{\mathfrak{R}_0}(b) = tp^{\mathfrak{A}}(H(b))$.
2. For every $a, b \in \mathfrak{R}_0, a \neq b$, put $tp^{\mathfrak{R}_0}(a,b) = \overline{tp^{\mathfrak{A}}(a), tp^{\mathfrak{A}}(b)}$.

After performing stage 0 condition (m1) holds since elements of L_0 were chosen in an appropriate way. Condition (m2) holds since the negative links $\overline{tp^{\mathfrak{A}}(a), tp^{\mathfrak{A}}(b)}$ are in $\mathcal{T}$ by part 2 of Proposition 5.7. Conditions (m3) - (m8) are obvious.

Stage k. $(k > 0)$ Put $\mathfrak{R}_k = \mathfrak{R}_{k-1}, L_k = \emptyset$.

1. For every $a \in L_{k-1}$, for every transitive predicate letter $T \in \sigma$, if a was not canceled with respect to T, then
   (a) *Creation of a T-petal of a:*
   Let $\mathfrak{D}$ be a T-petal of $[H(a)]^{\mathfrak{A}}_T$ and let G be the function given by Definition 5.8,
   find $d \in D$ such that $G(d) = H(a)$,
   put $\mathfrak{R}_k = \mathfrak{R}_k(a) \circ \mathfrak{D}(d)$ and add to L_k all the elements of D except d,
   for every $b \in D$, put $H(b) = G(b)$.
   (b) *Transitive closure for the T-petal of a:*
   For every $b \in D \setminus \{a\}$ and $c \in R_k \setminus D$, if $tp^{\mathfrak{R}_k}(c,a) \models T(x,y) \vee T(y,x)$ then put $tp^{\mathfrak{R}_k}(b,c) = \overleftrightarrow{tp^{\mathfrak{A}}(H(b), H(c)), T}$.
   (c) *Other types:*
   For every $b \in D$ and $c \in R_k \setminus D$, if $tp^{\mathfrak{R}_k}(b,c)$ is not defined, then put $tp^{\mathfrak{R}_k}(b,c) = \overline{tp^{\mathfrak{R}_k}(b), tp^{\mathfrak{R}_k}(c)}$ and cancel b with respect to T.
2. For every $a \in L_{k-1}$, for every $B \in \sigma$, for every conjunct $\gamma \in \Delta$ of the form (n2): $\forall x(\alpha(x) \rightarrow \exists y(\beta(x,y) \wedge \psi(x,y)))$ such that $\beta(x,y)$ contains B, if there is no witness of a for γ in $[a]^{\mathfrak{R}_k}_B$, then
   (a) *Witness of a for γ:*
   Find a witness d_γ of $H(a)$ for γ in $\mathfrak{A}$,
   add a new element b to L_k and put $H(b) = d_\gamma$,
   put $tp^{\mathfrak{R}_k}(a,b) = \overleftrightarrow{tp^{\mathfrak{A}}(H(a), H(b)), B}$.
   (b) *Transitive closure for the witness:*
   If B is a transitive predicate letter, say T, then for every $c \in R_k$, $c \neq a$, $c \neq b$, if either $tp^{\mathfrak{R}_k}(c,a) \models T(x,y)$ and $tp^{\mathfrak{R}_k}(a,b) \models T(x,y)$ or $tp^{\mathfrak{R}_k}(c,a) \models T(x,y)$ and $tp^{\mathfrak{R}_k}(a,b) \models T(x,y)$, then put $tp^{\mathfrak{R}_k}(b,c) = \overleftrightarrow{tp^{\mathfrak{A}}(H(b), H(c)), T}$.

(c) *Other types:*
For every $c \in R_k$, if $tp^{\mathfrak{R}_k}(b,c)$ is not defined, then put $tp^{\mathfrak{R}_k}(b,c) = \overrightarrow{tp^{\mathfrak{R}_k}(b), tp^{\mathfrak{R}_k}(c)}$.

Comments. (1b.) Note that $tp^{\mathfrak{R}_k}(b,c) = \overleftrightarrow{tp^{\mathfrak{A}}(H(b),H(c)),T}$ is well-defined since $H(b) \neq H(c)$. Towards a contradiction, assume $H(b) = H(c)$. Since $b \in \mathfrak{D}$, so, by condition (m5), $H(b) \in [H(a)]^{\mathfrak{A}}_T$. Then $H(c) \in [H(a)]^{\mathfrak{A}}_T$, so $tp^{\mathfrak{A}}(H(c),H(a)) \models T(x,y) \wedge T(y,x)$ and then $\overleftrightarrow{tp^{\mathfrak{A}}(H(c),H(a)),T}$ is T-single-transitive symmetric. Since a is not canceled then, by (m6), $tp^{\mathfrak{R}_k}(c,a)$ is single transitive oriented. But by (m5), $tp^{\mathfrak{R}_k}(c,a) = \overleftarrow{tp^{\mathfrak{A}}(H(c),H(a)),T}$, a contradiction.

(1c.) After performing this step condition (m7) holds. Additionally, (m8) holds, since in steps 1(b) and 1(c) only transitive-less or T-single-transitive oriented types are put.

Observe that after performing step 1 all the conditions (m1)-(m8), except condition (m3) hold.

(2a.) Note that by (m8), $[a]^{\mathfrak{A}}_B$ is a B-petal of $[H(a)]^{\mathfrak{A}}_B$. So, by condition 4 of definition 5.8, $d_\gamma \notin [H(a)]^{\mathfrak{A}}_B$ and so in case B is a transitive predicate letter, $tp^{\mathfrak{A}}(H(a),d_\gamma) \not\models B(x,y) \wedge B(y,x)$. Since $H(a) \neq H(b)$, so the type $\overleftrightarrow{tp^{\mathfrak{A}}(H(a),H(b)),B}$ is well defined and, in case B is a transitive predicate letter, it is a B-single-transitive oriented type, else it is a transitive-less type.

(2b.) Note that $\overleftrightarrow{tp^{\mathfrak{A}}(H(b),H(c)),T}$ is well-defined since $H(b) \neq H(c)$. Towards a contradiction, assume $H(b) = H(c)$. Assume, as one of two symmetric cases, $tp^{\mathfrak{R}_k}(c,a) \models T(x,y)$ and $tp^{\mathfrak{R}_k}(a,b) \models T(x,y)$. Then, by (m5), $tp^{\mathfrak{A}}(H(c),H(a)) \models T(x,y)$ and $tp^{\mathfrak{A}}(H(b),H(a)) \models T(y,x)$, so, $tp^{\mathfrak{A}}(H(b),H(a)) \models T(x,y) \wedge T(y,x)$. By (m5), $tp^{\mathfrak{R}_k}(b,a) = \overleftrightarrow{tp^{\mathfrak{A}}(H(b),H(a)),T}$, so $tp^{\mathfrak{R}_k}(a,b)$ is T-single-transitive symmetric which is a contradiction with the observation made in the previous step.

After performing this step T is transitive in $\mathfrak{R}_k$, since T was transitive in $\mathfrak{R}_k$ before performing step 2(b) and the pair $\langle b,c\rangle$ is in the transitive closure of T if and only if the pair $\langle a,c\rangle$ is in the transitive closure.

(2c.) Observe that conditions (m6) and (m8) hold, since in steps 2(a) - 2(c) only transitive-less or single-transitive oriented types are put.

After performing step 2 it is ensured that condition (m3) holds. So, by inductive hypothesis and by the comments given in steps 2(a) - 2(c), all the conditions (m1) - (m8) hold. ■

5.2 Flowers

In this section we introduce the notion of a *flower* which contains information about cliques and colors of elements non-symmetrically connected with a fixed element of a ramified model. We observe that the set of flowers realized in a ramified model for a [GF^2+TG]-sentence satisfies several properties that are collected in the definition of a *carpet*. We show in Theorem 5.13 that these properties are necessary and sufficient for satisfiability of a [GF^2+TG]-sentence.

Recall that m is the number of transitive predicate letters in σ and $M = \{1,\ldots,m\}$.

Definition 5.11 *A* flower F *is a triple* $F = \langle p^F, \{\mathfrak{D}^F_i\}_{i\in M}, \{In^F_i\}_{i\in M}\rangle$, *where*

(1) $\bigcap_{i\in M} D^F_i = \{p^F\}$,

(2) $\mathfrak{D}^F_i = [p^F]^{\mathfrak{D}^F_i}_{T_i}$ *and* $card(D^F_i) = 2^{O(card(\sigma))}$, *for every* $i \in M$,

(3) for every $i \in M$, *for every* $a,b \in D^F_i, a \neq b$, *the type* $tp^{\mathfrak{D}^F_i}(a,b)$ *is* T_i*-single-transitive,*

(4) In^F_i *is a set of 1-types, for every* $i \in M$.

The element p^F *is called a* pistil *of the flower* F, *the structures* $\mathfrak{D}^F_i$ *are* petals *of* F.

Note that it follows from the definition that the intersection of two distinct petals is a one-element set containing the pistil. We write $tp(p^F)$ to denote the type $tp^{\mathfrak{D}^F_i}(p^F)$ $(= tp^{\mathfrak{D}^F_i}(p^F))$.

Definition 5.12 *Let* Δ *be a* [GF^2+TG]*-sentence in normal form, let* $\mathcal{T}$ *be a set of types and let* $\mathcal{F}$ *be a set of flowers. We say that the pair* $\langle \mathcal{T}, \mathcal{F}\rangle$ *is a* Δ-carpet *if the following conditions hold:*

(c1) $\mathcal{T}$ *is* Δ*-acceptable and* $\mathcal{T} = \overline{\mathcal{T}}$,

(c2) *for every* $F \in \mathcal{F}$, *for every* $i \in M$, *we have*

(a) for every $a,b \in D^F_i$ *we have* $tp^{\mathfrak{D}^F_i}(a,b) \in \mathcal{T}$,

(b) for every $v \in In^F_i$ *there is a* T_i*-single-transitive oriented 2-type* $t \in \mathcal{T}$ *such that* $t \models tp(p^F)(x) \wedge v(y) \wedge T_i(y,x)$,

(c) for every $a \in D^F_i$ *there exists a flower* $W \in \mathcal{F}$ *such that* $\langle \mathfrak{D}^F_i, a\rangle \cong \langle \mathfrak{D}^W_i, p^W\rangle$ *and* $In^W_i = In^F_i$,

(c3) *for every conjunct* γ *of* Δ *of the form (n1):* $\exists x(\alpha(x) \wedge \psi(x))$ *there exists a flower* $F \in \mathcal{F}$ *such that* $tp(p^F) \models \alpha(x) \wedge \psi(x)$,

(c4) *for every* $F \in \mathcal{F}$, *for every* $i \in M$, *for every conjunct* γ *of* Δ *of the form (n2):* $\gamma = \forall x(\alpha(x) \rightarrow \exists y(\beta(x,y) \wedge \psi(x,y)))$, *if there is no witness of* p^F *for* γ *in any petal* $\mathfrak{D}^F_i$ *then there exists a flower* $W \in \mathcal{F}$ *and a 2-type* $t \in \mathcal{T}$ *such that*

(a) $t \models tp(p^F)(x) \wedge tp(p^W)(y) \wedge \beta(x,y) \wedge \psi(x,y)$,

(b) *if* $\beta = T_i(y,x)$ *then*

i. t *is* T_i*-single-transitive oriented,*
ii. $tp(p^W) \in In_i^F$,
iii. $In_i^F \supseteq In_i^W \cup \{tp^{\mathfrak{D}_i^W}(d) : d \in \mathfrak{D}_i^W\}$,

(c) *if* $\beta = T_i(x,y)$ *then*

i. t *is* T_i*-single-transitive oriented,*
ii. $tp(p^F) \in In_i^W$,
iii. $In_i^W \supseteq In_i^F \cup \{tp^{\mathfrak{D}_i^F}(d) : d \in \mathfrak{D}_i^F\}$,

(d) *if* β *does not contain a transitive predicate letter then* t *is transitive-less.*

Theorem 5.13 *A* [GF^2+TG]*-sentence* Δ *in normal form is satisfiable if and only if there exists a* Δ*-carpet.*

5.3 Complexity

Theorem 5.14 *The satisfiability problem for* [GF^2+TG] *is in* 2EXPTIME.

Proof Let Γ be a [GF^2+TG]-sentence over a signature τ and let n be the length of Γ. Let $\mathcal{D}$ be the set of [GF^2+TG]-sentences in normal form over a signature σ given by Lemma 3.2. Then, Γ is satisfiable if and only if there exists a satisfiable sentence $\Delta \in \mathcal{D}$. Moreover, $card(\sigma) = O(n)$ and the length of Δ is linear with respect to n.

By Theorem 5.12, a sentence $\Delta \in \mathcal{D}$ is satisfiable if and only if there exist a set of 1- and 2-types $\mathcal{T}$ and a set of flowers $\mathcal{F}$ such that $\langle \mathcal{T}, \mathcal{F} \rangle$ is a Δ-carpet.

Every type of the set $\mathcal{T}$ can be written using $O(n)$ space and $card(\mathcal{T}) \leq 2^{4n \log n}$. Define $N(\Delta)$ as the number of all flowers. Since every flower can be written using $2^{cn \log n}$ space, $N(\Delta) \leq 2^{2^{cn \log n}}$, for a constant c.

The following alternating exponentially space-bounded algorithm is a satisfiability test for [GF^2+TG]- sentences.

Input: a [GF^2+TG]-sentence Γ;
Compute the set $\mathcal{D}$;
guess a sentence $\Delta \in \mathcal{D}$;
Compute $N(\Delta)$;
guess a set of types $\mathcal{T}$;
if $\mathcal{T}$ does not satisfy condition (c1) then reject;
universally choose a conjunct $\gamma \in \Delta$ of the form
(n1) $\exists x(\alpha(x) \wedge \psi(x))$;
guess a 1-type $t \in \mathcal{T}$ and a flower F;
if $tp(p^F) \neq t$ or $t \not\models \alpha(x) \wedge \psi(x)$ then reject;
for $j = 1$ to $N(\Delta)$ do
universally choose the Case:
Case 1 *Condition (c2)*
universally choose $i \in M$;
if F does not satisfy (c2a)-(c2b) then reject;
universally choose $a \in D_i^F$;
guess a flower W;
if $\langle \mathfrak{D}_i^F, a \rangle \not\cong \langle \mathfrak{D}_i^W, p^W \rangle$ or $In_i^W \neq In_i^F$
then reject;
F:= W;
Case 2 *Condition (c4)*
universally choose $i \in M$ and a conjunct γ of Δ of the form
(n2) $\forall x(\alpha(x) \rightarrow \exists y(\beta(x,y) \wedge \psi(x,y)))$;
if there is no $b \in D_i^F$ such that
$\mathfrak{D}_i \models \alpha(p^F) \rightarrow \beta(p^F, b) \wedge \psi(p^F, b)$ then
begin
guess a flower W and a 2-type $t \in \mathcal{T}$;
if conditions (c4a) - (c4d) do not hold
then reject;
end;
F:=W;
od;
accept

It is well known (see [3]) that for all functions $f(n) \geq \log n$, ASPACE($f(n)$) = $\bigcup_{c \in \mathbb{N}}$ TIME($2^{cf(n)}$). In particular AEXPSPACE = 2EXPTIME. ■

6 The general case

The ideas and methods used for [GF^2+TG] can be extended to obtain the analogous results for the whole [GF+TG].

References

[1] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *ILLC Research Report ML-1996-03*, University of Amsterdam, 1996. Journal version in: *J. Philos. Logic*, 27 (1998), no. 3, 217-274.

[2] F. Baader, H. Bürckert, B. Hollunder, A. Laux, and W. Nutt. Terminologische logiken. *KI*, 3/92:23–33, 1992.

[3] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II.* Springer, 1990.

[4] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82:353–367, 1996.

[5] R. Brafman, J. C. Latombe, Y. Moses, and Y. Shoham. Knowledge as a toll in motion planning under uncertainty. In *R. Fagin, ed., 'Theoretical Aspects of Reasoning about Knowledge: Proc.*

Fifth Conference', pages 208–224. Morgan Kaufmann, 1994.

[6] M. Burrows, M. Abadi, and R. Needham. Authentication: a practical study in belief and action. In *Proc. Second Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 325–342, 1988.

[7] J. M. V. Castilho, M. A. Casanova, and A. L. Furtado. A temporal framework for database specification. In *Proc. Eigth Int. Conf. on Very Large Data Bases*, pages 280–291, 1982.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131, pages 52–71. Springer-Verlag, 1981.

[9] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. The MIT Press, Cambridge, MA, 1990.

[10] D. Gabbay. Expressive functional completeness in tense logic. In *U. Mönnich, ed., 'Aspects of Philosophical Logic'*, pages 91–117. Reidel, 1971.

[11] H. Ganzinger, C. Meyer, and M. Veanes. The two-variable guarded fragment with transitive relations. In *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 24–34, 1999.

[12] E. Grädel. On the restraining power of guards. *J. Symbolic Logic*, 64:1719–1742, 1999.

[13] E. Grädel, P. Kolaitis, and M. Vardi. On the decision problem for two-variable first-order logic. *Bull. of Symb. Logic*, 3(1):53–69, 1997.

[14] E. Grädel, M. Otto, and E. Rosen. Undecidability results on two-variable logic. In *14th Symposium on Theoretical Aspects of Computer Science*, volume LNCS 1200, pages 249–260, 1997. to appear in Archive for Mathematical Logic.

[15] E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 45–54, 1999.

[16] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 32:137–161, 1985.

[17] N. Immerman and M. Y. Vardi. Model checking and transitive-closure logic. In *Lecture Notes in Computer Science*, volume 1254, pages 291–302. Springer Verlag, 1997.

[18] E. Kieroński. *private communication*, 2000.

[19] H. R. Lewis. Complexity results for classes of quantificational formulas. *J. Comp. and System Sci.*, 21:317–353, 1980.

[20] W. Lipski. On the logic of incomplete information. In *Proc. Sixth Int. Symp. on Mathematical Foundations of Computer Science*, volume 53, pages 374–381. Springer Verlag, 1977.

[21] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *D. Michie, ed. 'Machine Intelligence'*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[22] M. Mortimer. On languages with two variables. *Zeitschr. f. Logik und Grundlagen d. Math.*, 21:135–140, 1975.

[23] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundations of Computer Science*, pages 46 57, 1977.

[24] V. R. Pratt. Semantical considerations on floyd-hoare logic. In *Proc. 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[25] D. Scott. A decision method for validity of sentences in two variables. *J. Symb. Logic*, 27:477, 1962.

[26] W. Szwast and L. Tendera. On the complexity of the monadic guarded fragment with transitivity. unpublished manuscript.

[27] J. van Benthem. Dynamics bits and pieces. *ILLC Research Report LP-97-01*, University of Amsterdam, 1997.

[28] M. Y. Vardi. Why is modal logic so robustly decidable? *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*,, 31:149–184, 1997.

The hierarchy inside closed monadic Σ_1 collapses on the infinite binary tree

Andre Arnold Giacomo Lenzi* Jerzy Marcinkowski†

Laboratoire Bordelais de Recherche en Informatique, Université de Bordeaux I, 351, cours de la Libération, 33 405 Talence cedex, France
{arnold,lenzi}@labri.u-bordeaux.fr

Institute of Computer Science, University of Wrocław, Przesmyckiego 20, 51151 Wrocław, Poland,
jma@tcs.uni.wroc.pl

Abstract

Closed monadic Σ_1, as proposed in [AFS98], is the existential monadic second order logic where alternation between existential monadic second order quantifiers and first order quantifiers is allowed. Despite some effort very little is known about the expressive power of this logic on finite structures. We construct a tree automaton which exactly characterizes closed monadic Σ_1 on the Rabin tree and give a full analysis of the expressive power of closed monadic Σ_1 in this context. In particular, we prove that the hierarchy inside closed monadic Σ_1, defined by the number of alternations between blocks of first order quantifiers and blocks of existential monadic second order quantifiers collapses, on the infinite tree, to the level 2.

1 Introduction

The monadic second order logic (MSOL) has long been studied by computer scientists in at least two contexts: descriptive complexity on finite structures (the Fagin context) and theory of finite automata on infinite trees (the Rabin context). Although the results of this paper concern infinite trees, rather than the finite structures, the class considered in this paper (the closed Σ_1 hierarchy) originates from the descriptive complexity area.

1.1 Previous works

In the **Fagin context**, the expressive power of MSOL on finite structures is studied. The motivation comes from the fact that a property of finite structures is in the class NP if and only if it is expressible by an existential second order sentence ([F74]). The question if NP equals co-NP is in this way reduced to the one if Σ_1, the set of properties expressible by existential second order sentences equals to Π_1, the set of properties expressible by universal second order sentences. But the last question is far beyond the techniques we have. That is why, as suggested by Fagin ([F75]) we first study the monadic counterparts of the complexity questions. The monadic NP is the set of properties expressible by a formula with the quantifier prefix of the form $\exists^*(\forall\exists)^*$ (where $\exists$ and $\forall$ are monadic second order quantifiers while $\exists$ and $\forall$ are first order quantifiers). It is not very hard to prove ([F75]) that monadic NP does not equal monadic co-NP: graph connectivity belongs to the second of these classes but not to the first. In last decade a number of deep techniques was developed to show non-expressibility results: in [S94] Schwentick developed sophisticated strategies for Ehrenfeucht-Fraïssé games to prove that graph connectivity is not in monadic NP even in the presence of a built-in order. In [AF90] Ajtai and Fagin used complicated probabilistic argument to show that reachability in directed graphs is not in monadic NP (also [AF97]). Finally, Matz and Thomas ([MT97]) constructed an automata theory based proof of the theorem that monadic hierarchy, the counterpart of Stockmeyer polynomial hierarchy, is strict. The last means that for every natural n there is a property of finite structures expressible by a formula with quantifier prefix of the form $(\exists^*\forall^*)^{n+1}(\exists\forall)^*$ but not by one with $(\exists^*\forall^*)^n(\exists\forall)^*$. Their proof, as we said, is automata theory based. Let us sketch it here. The structures they consider are colored rectangular grids. They treat such a grid as a word of columns. So each property of grids can also be viewed as set of words. For each formula ϕ of monadic second order logic they construct a finite word automaton $A(\phi)$ which recognizes exactly the set of colored grids in which ϕ is valid. It is easy to construct $A(\exists\phi)$: the set of states remains the same as in $A(\phi)$ and the transition function becomes "more nondeterministic". But it turns out that $A(\forall\phi)$ can only be constructed as $A(\neg\exists\neg\phi)$, which requires computing a complement of the original automa-

*Supported by an Italian CNR grant
†Supported by Polish KBN grant 2 PO3A 018 18

0-7695-1281-X/01 $10.00 © 2001 IEEE

ton (via determinisation) and leads to exponential explosion of the number of states. Then they use the number of states argument, and the fact that they are considering 2-dimensional grids rather than words, to get the separation result.

It could appear at this stage that we were able to answer every interesting question about the Fagin world. But in [AFS98],[AFS00] Ajtai, Fagin and Stockmeyer noticed that monadic NP is not the most general monadic subclass of NP: they defined *closed monadic NP* as the class of properties definable by a formula with the prefix of the form $(\exists^*(\exists\forall)^*)^*$. In their paper they prove that such possibility of alternation between first order quantifiers and monadic second order existential quantifiers increases the expressive power of the language. Closed monadic NP does not share the obvious pathologies of monadic NP: connectivity and directed reachability are definable now, the first of them by a prefix of the form $\forall\forall\exists^*(\forall\exists)^*$, the second by $\exists\forall\exists(\forall\exists)^*$. To show that the increase of expressive power is indeed substantial the authors define a property $\mathcal{P}_2$, definable by $\exists^*(\exists\forall)^*\exists^*(\exists\forall)^*$ but (and this proof is the main technical contribution of [AFS98]) not by any boolean combination of formulae with the prefix of the form $(\forall\exists)^*\exists^*(\forall\exists)^*$. Some natural questions are stated in the paper: is closed monadic NP equal to closed monadic co-NP ? Does there exist any property in monadic hierarchy but not in closed monadic NP ? Another very natural question one can ask here is if the hierarchy inside closed monadic NP defined by the number of blocks of monadic second order quantifiers is strict (the n-th level of the hierarchy are here the properties definable by $(\exists^*(\exists\forall)^*)^n$).

All the above questions seem to be hard. It is amazing, but despite some effort ([AFS00],[M99],[JM01]) we are not even able to show that there is any property in monadic hierarchy but not on level 2 of the hierarchy inside closed monadic NP, (i.e. not definable by a formula with the prefix of the form $\exists^*(\exists\forall)^*\exists^*(\exists\forall)^*$). One could think that a positive answer to the last questions should follow from the Matz-Thomas technique, but this is not the case: it turns out that the construction of the finite automaton for $\forall\phi$ leads to the same kind of exponential state explosion as the construction of $A(\forall\phi)$. The property expressible with a prefix $(\exists^*\forall^*)^{n+1}(\exists\forall)^*$ but not with $(\exists^*\forall^*)^n(\exists\forall)^*$ constructed with the Matz-Thomas method is in fact expressible also by a formula with the prefix $\exists^*(\exists^*\forall^*)^{n+1}\exists^*(\exists\forall)^*$, which means that it is on the second level of the hierarchy inside closed monadic NP. It seems possible that understanding the difference between the increase of the complexity of a recognizable language induced by first order universal quantification and by monadic second order universal quantification may lead to the answer to some of the open questions from [AFS98]. In this paper we show that at least in the Rabin context all those questions can be answered in this way.

In the **Rabin context**, one considers finite automata and MSOL on infinite trees. The subject was pioneered by Rabin in his seminal paper [R69], where he proves that MSOL over the infinite binary tree (also known as $S2S$, the monadic second order theory of two successors) collapses to Σ_3 and is decidable. This deep result has been later used as a tool to solve many other decision problems by reduction to $S2S$.

The main tool of Rabin's proof are what we call today Rabin automata, which characterize MSOL on Rabin trees. Rabin's proof is difficult to understand, and in particular the proof that Rabin automata are closed under complement is very complicated; so, many papers have been devoted to explain what is really going on in the complementation proof. Many sorts of tree automata have been defined in order to simplify the construction including Muller automata (originally considered for infinite words [M63]), and Streett automata, [S82]. But the real progress came only as late as in [GH82] with the introduction of games on trees. Games correspond to automata and they are easy to complement (by determinacy). A related concept are alternating tree automata, which generalize usual, nondeterministic automata [MS87]: an accepting run of such an automaton looks much like a winning strategy in a game. After thirty years of studies there is still research under way which gives us better understanding of tree automata, for instance in [A94] a fully "algebraic" proof of the complementation lemma is given, in [Z98] the author proposes a proof of the lemma via infinite games on graphs; and [EJ99], where the complexity of the satisfiability problem on tree automata is investigated.

A very natural class of automata traditionally studied in this context are Büchi automata, originally introduced in [B60] for infinite words. The expressive power of automata of this class on the infinite tree is extensively studied in Rabin's paper [R70]. Among other things, Rabin shows that Büchi definable sets form a proper subclass of MSOL definable sets, and that they have a number of closure properties (including closure under weak universal quantification). One of the closure properties proved by Rabin (Theorem 9 in [R70]) gave us some inspiration for the construction performed in the present paper. Büchi automata, despite of their simple definition which makes them a much nicer object of studies than Rabin automata, are also not fully understood yet. For example, only recently in [L01], it has been shown that Büchi automata express exactly the properties in Σ_2 on the binary tree.

1.2 Our contribution

We give complete analysis of the structure of closed monadic NP (which here should be rather called closed monadic Σ_1) in the world of Rabin. Our original goal was to look for techniques that would establish the strictness of

the hierarchy inside closed monadic Σ_1 in this context.

Our first result was the one from Appendix 3 that the property EGFP (the last is a CTL* formula saying that there is a path on which the predicate P occurs infinitely many times) is expressible by a formula with the prefix of the form $\exists^*(\exists\forall)^*\ \exists^*(\exists\forall)^*$ but not by a Σ_1 formula. But it turns out that EGFP is even harder: as we prove in Section 2.4 it is not expressible by any boolean combination of formulae with the prefix of the form $(\forall\exists)^*\ \exists^*(\forall\exists)^*$ and in this way it is a counterpart, in the Rabin world, of the property $\mathcal{P}_2$ from [AFS98].

Then we tried different kinds of nesting of CTL* formulae to construct properties in closed monadic Σ_1 but not on its second level. All the attempts failed: to our great surprise the hierarchy inside closed monadic Σ_1 collapses in the Rabin world. As we prove in Section 3 it collapses to level 2, the same mysterious level 2 that we are not able to move beyond in the Fagin context. Our proof technique is based on automata. We define a kind of a finite tree automaton which we call *search automaton* (to guess how such an automaton should be defined was the most difficult part of our research). Search automaton is a kind of Büchi automaton with additional simple requirements on accepting conditions. The class of languages recognizable by search automata contains Σ_1 and is closed under existential monadic second order quantification (this is not hard to prove) and under universal first order quantification. But the class is not closed under universal monadic second order quantification! In Section 2.3 we show that the property AFP (on every path there occurs P), clearly expressible by a formula of monadic second order logic with the quantifier prefix of the form $\forall(\forall\exists)^*$ is not recognizable by any search automaton and thus not in closed monadic Σ_1. This means that our search automata, unlike Rabin automata, and unlike finite automata on words are sensitive to the difference between first and monadic second order universal quantification.

In Section 2.1 we show that if a property is recognizable by a search automaton then it is expressible by a Σ_1 formula using an additional predicate $<$, where the meaning of $x < y$ is *x is a prefix of y*. The last formula, for fixed x and y can be defined by additional existentially quantified monadic relation, and as a result we get a formula with the prefix of the form $\exists^*(\exists\forall)^*\ \exists^*(\exists\forall)^*$.

1.3 Remark on $\Sigma_1(TC)$

There is another unexpected link between our two worlds: not only, as we said in Subsection 1.1, no monadic property in the Fagin world is known which could be proved not to be expressible on the level two of the hierarchy inside closed monadic NP. In fact we do not even know a monadic property provably not in $\Sigma_1(TC)$[1] where TC is the simplest possible version of the transitive closure operator: $TC(\phi, x, y)$, where ϕ is a first order formula with two free variables, means that there is a finite path $x = x_1, x_2, \ldots x_k = y$ such that $\phi(x_i, x_i + 1)$ holds for each i. The formula ϕ is generalized graph edge, and it is as hard to define the TC operator as to define graph reachability. This means that all properties in $\Sigma_1(TC)$ are, in the Fagin world, definable by a formula with a prefix of the form $(\exists^*(\exists\forall)^*)^3$. Could it be possible that closed monadic NP in the Fagin world is exactly $\Sigma_1(TC)$? It follows from our results that the classes are equal in the Rabin world.

2 Technical part, the easy fragment

We consider MSOL over trees. By a *tree* we mean a structure whose domain is $\{0,1\}^*$. The signature consists of two functions *left son* and *right son*, mapping each w to $w0$ and $w1$ respectively, and of some finite set $\mathcal{P}$ of monadic predicates. Sometimes (when explicitly mentioned) the signature will also contain the prefix ordering relation $<$, with the natural meaning. When talking about automata we often think that a tree is rather a function from $\{0,1\}^*$ to some alphabet X than a structure. Since X can be viewed as the powerset of $\mathcal{P}$ the two definitions are equivalent. A *path* in a tree is an infinite sequence $x_1, x_2 \ldots$ of words from $\{0,1\}^*$ such that each x_{i+1} equals to $x_i 0$ or to $x_i 1$.

2.1 Search automata

A search automaton over an alphabet X is a tuple $A = \langle Q, Q_0, R, \{F_1, F_2, \ldots F_k\}\rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ and R is a finite set of rules which are constructs of the form $(q, a) \to (q_0, q_1)$, with $q, q_0, q_1 \in Q$ and $a \in X$. F_i are subsets of Q and satisfy the following condition:

(L) For any rule $(q, a) \to (q_0, q_1)$ from R and for any $i : 1 \leq i \leq k$, if $q \notin F_i$, then there exists $d \in \{0,1\}$ such that $q_d \in F_i$.

For a tree T with alphabet X a run of the automaton A on T is such a function $\rho : \{0,1\}^* \to Q$ that for each $w \in \{0,1\}^*$ $(\rho(w), T(w)) \to (\rho(w0), \rho(w1))$ is a rule from R. A run ρ is initial if $\rho(\epsilon) \in Q_0$ and is accepting if:

(C) for any path $w_1, w_2, \ldots$ and any i, $\rho(w_j) \in F_i$ for infinitely many numbers j.

We say that A recognizes the tree T if there exists an initial accepting run of A on T.

[1] The properties constructed by Matz-Thomas method are also in this class

Lemma 2.1 *If A is a search automaton, then the set of trees recognized by A is definable by a formula in $\Sigma_1(<)$.*

Proof: In presence of the condition (L), it is easy to see that (C) is equivalent to the first order condition:

(C') For any i and any $u \in \{0,1\}^*$ such that $\rho(u) \notin F_i$ there exists $v \geq u$ such that $\rho(v0) \in F_i$, $\rho(v1) \in F_i$ and $\forall w \in \{0,1\}^*, (u \leq w \leq v \rightarrow \rho(w) \notin F_i$.

In order to prove this equivalence consider, for given i and for given $u \in \{0,1\}^*$ such that $\rho(u) \notin F_i$, the set $A = \{w \in \{0,1\}^* : u \leq w \wedge \rho(w) \notin F_i \wedge \forall u \leq v \leq w \ \ \rho(v) \notin F_i\}$. By the condition (L) the set A is a path (finite or not). This path is finite if and only if it contains a word v such that $\rho(v0) \in F_i$ and $\rho(v1) \in F_i$. And the condition (C) means that for every u and every i the set A is finite. ■

2.2 Main result

Theorem 2.2 *Consider the following six classes of properties of trees:*

(i) Closed monadic Σ_1 (i.e. the set of properties definable by a formula of MSOL with the quantifier prefix of the form ($\exists^*(\forall\exists)^*)^*$).

(ii) Second level of closed monadic Σ_1 (i.e. the set of properties definable by a formula of MSOL with the quantifier prefix of the form $\exists^*(\forall\exists)^* \exists^*(\forall\exists)^*$).

(iii) Monadic $\Sigma_1(<)$ (i.e. the set of properties definable with the use of prefix ordering relation $<$ by a formula of MSOL with the quantifier prefix of the form $\exists^*(\forall\exists)^*$).

(iv) The properties recognizable by search automata.

(v) Monadic Π_1 (i.e. the set of properties definable by a formula of MSOL with the quantifier prefix of the form $\forall^*(\forall\exists)^*$).

(vi) First order closure of Σ_1 (i.e. the set of properties definable by a boolean combination of formulae of MSOL with the quantifier prefix of the form $(\forall\exists)^* \exists^*(\forall\exists)^*$

Then: (i)=(ii)=(iii)=(iv) and (v)$\neq$ (iv) and (vi) $\neq$ (ii).

Proof: (v) $\neq$ (iv) is proved in Section 2.3, (vi)$\neq$ (ii) is proved in Section 2.4, (ii)$\subseteq$ (i) is obvious, also (iii)$\subseteq$ (ii) is easy to show, (iv)$\subseteq$ (iii) was proved in the previous Section. For the proof of (i)$\subseteq$ (iv) see Section 3 ■

Let us remark that it follows from Theorem 2.2 that closed monadic $\Sigma_1(<)$ on the Rabin tree is exactly monadic $\Sigma_1(<)$, the possibility of alternating first and second order quantifiers does not give any additional power here.

2.3 AFP is not in closed monadic Σ_1

Lemma 2.3 *Let P be a monadic predicate. The property that on every infinite path from the root there is a P (defined by the CTL formula AFP), is definable by a formula with quantifier prefix of the form $\forall^*(\exists\forall)^*$ (is in monadic Π_1) but is not recognizable by a search automaton.*

Proof: Suppose A were a search automaton for AFP, over the alphabet $\{P, \neg P\}$, with a set Q of states and k accepting conditions $F_1, \ldots, F_k$. Let $l = |Q|$. Then consider a natural number m which is sufficiently large, say $m = 2(l+1)k + 2$. Let M be the tree where P is interpreted as the set of all words of length m. Obviously, M has the property AFP. Let ρ be an accepting run of A on M. Since A is a search automaton, there exists a sequence of $(l+1)k$ words:

$$w_{1,1} < \ldots < w_{1,k} < \ldots < w_{l+1,1} < \ldots < w_{l+1,k}$$

such that for each pair i, j it holds that $w_{i,j}$ has length less than m, that $\rho(w_{i,j}) \in F_j$, and that the distance between each two consecutive elements of the sequence is 1 or 2. Now, we find $i < i' \leq l+1$ such that $\rho(w_{i,1}) = \rho(w_{i',1})$ and apply a "pumping" argument: let M' be a tree with predicate P defined as:

(i) If $w_{i,1}$ is not a prefix of w then $w \in P$ holds in M' if and only if it holds in M.

(ii) Let y be such that $w_{i,1}y = w_{i',1}$. If w is of the form $w_{i,1}y^*x$ where y is not a prefix of x then $w \in P$ holds in M' if and only if $w_{i,1}x \in P$ holds in M.

In an analogous way define the run ρ' of A on M'. Then M' does not have the property AFP: words being prefixes of some word of the form $w_{i,1}y^*$ form an infinite path without P. But ρ' is an accepting run of A on M'. ■

2.4 Monadic Σ_1 is much less than closed monadic Σ_1

Let us consider the following property of infinite binary trees colored with a monadic relation P:
(*) there is an infinite sequence $x_1, x_2, x_3 \ldots$ of vertices such that $x_i < x_{i+1}$ and $P(x_i)$ hold for each i .

In the notation of the temporal logic CTL* the property (*) can be expressed as EGFP.

It is easy to see that (*) is expressible in monadic $\Sigma_1(<)$. Our original elementary proof of the fact that EGFP is not in monadic Σ_1 can be found in the Appendix. But it turns out that with the use of a new result from [L01] we can easily have something more:

Theorem 2.4 *Property $(*)$ is not expressible by any boolean combination of formulae with the prefix of the form $(\forall\exists)^* \mathbf{\exists}^*(\forall\exists)^*$.*

Proof: It is easy to see that if some property is expressible by any boolean combination of formulae with the prefix of the form $(\forall\exists)^* \mathbf{\exists}^*(\forall\exists)^*$ then it is also expressible by a formula with the prefix of the form $(\forall\exists)^* \mathbf{\exists}^* \mathbf{\forall}^*(\forall\exists)^*$. It is proved in [L01] that if a property is in monadic Σ_2, that is expressible by a formula with the prefix $\mathbf{\exists}^* \mathbf{\forall}^*(\forall\exists)^*$, then it is recognizable by a Büchi automaton. But the class of properties recognizable by Büchi automata is closed under universal first order quantification ([R70]) and under existential first order quantification. So if a property is expressible by a boolean combination of formulae with the prefix of the form $(\forall\exists)^* \mathbf{\exists}^*(\forall\exists)^*$ then it is recognizable by a Büchi automaton.

If property (*) were expressible by a formula like in the theorem then its complement:

(**) On each path there are only finitely many P

would also be expressible in this class, and thus would be Büchi. But the last is not the case as proved in [R70]. ■

3 Technical part, the harder fragment

In this technical section we prove that search automata recognize all the properties in closed monadic Σ_1. We leave it for the reader as an easy exercise to prove that they recognize all the properties in monadic Σ_1, and that the class of recognizable properties is closed under existential quantification, first order and monadic second order. What remains to be proved is:

Lemma 3.1 *Let A be a search automaton, over some alphabet $\Sigma \times \{x, \bar{x}\}$. For a given tree T over Σ and for $v \in \{0,1\}^*$ define T^v as the tree over $\Sigma \times \{x, \bar{x}\}$, with $T^v(y) = \langle T(v), x\rangle$ if $v = y$ and $T^v(y) = \langle T(v), \bar{x}\rangle$ otherwise. Then there exists a search automaton $\forall A$ over the alphabet Σ such that $\forall A$ accepts a tree T if and only if A accepts every T^v for $v \in \{0,1\}^*$.*

From now on $A = \langle Q, Q_0, R, \{F_1, F_2, \ldots F_k\}\rangle$ is a fixed search automaton.

3.1 Multiruns

Let us assume that the tree T over Σ is such that each T^v is recognized by A. Then for each v there is an initial accepting run ρ^v on T^v. For any v and w in $\{0,1\}^*$ such that $w \not\leq v$, the subtree T^v_w of T^v defined by $T^v_w(u) = T^v(wu)$ is over the alphabet $\Sigma \times \{\bar{x}\}$ (more precisely, $T^v_w(u) = \langle T(wu), \bar{x}\rangle$), and ρ^v induces an accepting run ρ^v_w of A on T^v_w defined by $\rho^v_w(u) = \rho^v(wu)$.

The initial accepting runs of $\forall A$ on T must allow us to retrieve a family $(\rho^v)_{v\in\{0,1\}^*}$. In particular, they must contain in an encoded form, a sufficiently large set of runs ρ^v_w, called *multirun*.

Definition 3.2 *For a tree T over Σ we define $\bar{T}$ as the tree over $\Sigma \times \{\bar{x}\}$ with $\bar{T}(w) = \langle T(w), \bar{x}\rangle$. If T is a tree and $u \in \{0,1\}^*$, then we define T_u as the "subtree of T rooted in u": $T_u(y) = T(uy)$ for each y.*

Definition 3.3 *Let T be a tree over Σ. A* multirun *of the automaton A on T is a partial function Ψ whose arguments are pairs $\langle w, q\rangle$ (where $w \in \{0,1\}^*$ and $q \in Q$), such that for each element $\langle w, q\rangle$ of its domain $\Psi(w, q)$ is a run of A on $\bar{T}_w$, with the state q in the root w of $\bar{T}_w$. A multirun Ψ is* accepting *if all the runs $\Psi(w, q)$ are accepting.*

For convenience, instead of considering $\Psi(w, q)$, which is a total mapping, we consider the partial mapping of domain $w\{0,1\}^*$, denoted by $\Psi_{w,q}$ and defined by $\Psi_{w,q}(wu) = \Psi(w, q)(u)$, so that if $\Psi(w, q)$ is ρ^v_w then

(E) $\Psi_{w,q}(wu) = \Psi(w, q)(u) = \rho^v_w(u) = \rho^v(wu)$.

Multiruns are a way of remembering the whole interesting knowledge about possible accepting runs of A on such subtrees of $T \times \{x, \bar{x}\}$ which do not contain the (universally quantified) variable x. This interesting information is where we can start such a run and in which state we can start it:

Definition 3.4 *Two multiruns on T are* equivalent *if they have the same domain.*

We want to store the information as a run of our new automaton $\forall A$. But $\forall A$ can only store finite piece of information in each of the nodes of $\{0,1\}^*$. This motivates:

Definition 3.5 *A multirun Ψ is* uniform *if for each $w_1, w_2, y, z \in \{0,1\}^*$ and $q_1, q_2 \in Q$ the following implication holds:*
$(w_1 \leq y \leq z) \wedge (w_2 \leq y) \wedge \Psi_{w_1,q_1}(y) = \Psi_{w_2,q_2}(y)$ *implies* $\Psi_{w_1,q_1}(z) = \Psi_{w_2,q_2}(z)$

So a multirun is uniform, if each time when two of its runs agree on some node y they remain equal on all the successors of y.

Of course not every multirun is uniform, but it turns out that:

Lemma 3.6 *For every accepting multirun Ψ on T there exists an equivalent accepting multirun Ψ' which is uniform.*

The tool we need to prove Lemma 3.6 is:

Lemma 3.7 *Let T be a tree over Σ and let ρ be an accepting run of A on some $\bar{T}_w$. Let B be an antichain (with respect to the prefix ordering $\leq$) of nodes of T_w, and for each $y \in B$ let ρ_y be an accepting run of A on some subtree of $\bar{T}$ containing $\bar{T}_y$. Suppose for each $y \in B$ it holds that $\rho_y(y) = \rho(y)$. Then the function ϱ defined as:*

(i) $\varrho(z) = \rho_y(z)$ *if* $y \in B$ *and* $z \geq y$,

(ii) $\varrho(z) = \rho(z)$ *if* $z \in T_w$ *but* $z \notin T_y$ *for any* $y \in B$,

is an accepting run of A *on* $\bar{T}_w$. ■

Now, let Ψ be an accepting multirun. Let $\prec$ be any fixed total order on Q. We define the following ordering $\sqsubset$ on $dom(\Psi)$:

(i) $\langle w, q\rangle \sqsubset \langle w', q'\rangle$ if $|w| \langle |w'|$,

(ii) $\langle w, q\rangle \sqsubset \langle w', q'\rangle$ if $|w| = |w'|$ and w is smaller than w' in the lexicographic ordering of words,

(iii) $\langle w, q\rangle \sqsubset \langle w, q'\rangle$ if $q \prec q'$.

Obviously, $\sqsubset$ is a total order and if $dom(\Psi)$ is infinite then the type of $\sqsubset$ is ω. Now, we define a multirun Ψ' by induction on $\sqsubset$. Suppose, for some pair $\langle w, q\rangle \in dom(\Psi)$ the runs $\Psi'_{w',q'}$ are already defined for all pairs $\langle w'q'\rangle \sqsubset \langle w, q\rangle$. Let V be the set of such nodes z of T_w that there exists $\langle w', q'\rangle \sqsubset \langle w, q\rangle$ such that $\Psi'_{w',q'}(z) = \Psi_{w,q}(z)$, and let B be the set of minimal elements of V with respect to the prefix ordering $\leq$. Then, by Lemma 3.7 the function $\Psi'_{w,q}$ defined as:

(i) $\Psi'_{w,q}(z) = \Psi_{w',q'}(z)$ if $y \in B$ and $\Psi'_{w',q'}(y) = \Psi_{w,q}(y)$ and $z \geq y$

(ii) $\Psi'_{w,q}(z) = \Psi_{w,q}(z)$ if $z \in T_w$ but $z \notin T_y$ for any $y \in B$,

is an accepting run of A on T_w. It is also easy to see that $\Psi'_{w,q}(w) = q$, so Ψ' as we just defined it is an accepting multirun and is equivalent to Ψ. We leave it as an exercise for the reader to show that Ψ' is indeed uniform. ■

In order to construct a family $(\rho^v)_{v\in\{0,1\}^*}$ it is not enough to know a multirun. This is because only the values $\rho^v(w)$ where $w \not\leq v$ are kept in a multirun. To get the values of $\rho^v(w)$ where $w \leq v$, we need an additional piece of information:

Definition 3.8 *For a given multirun* Ψ *its* co-multirun, *which will be denoted as* $co\Psi$ *is a function with domain* $\{0,1\}^*$, *whose values are subsets of* Q, *defined by induction as:*

(i) $co\Psi(\varepsilon) = Q_0$;

(ii) $q_0 \in co\Psi(w0)$ *if and only if there exist* $q \in co\Psi(w)$ *and* $q_1 \in Q$ *such that* $\langle w1, q_1\rangle \in dom(\Psi)$ *and* $(q, \langle T(w), \bar{x}\rangle) \to (q_0, q_1)$ *is a rule of* A;

(iii) $q_1 \in co\Psi(w1)$ *if and only if there exist* $q \in co\Psi(w)$ *and* $q_0 \in Q$ *such that* $\langle w0, q_0\rangle \in dom(\Psi)$ *and* $(q, \langle T(w), \bar{x}\rangle) \to (q_0, q_1)$ *is a rule of* A.

Notice that $co\Psi$ only depends on the domain of Ψ: if Ψ and Ψ' are equivalent then $co\Psi$ equals to $co\Psi'$. So, by Lemma 3.6 for every multirun there exists a uniform one with the same co-multirun.

The following lemma says that the information carried by a multirun and its co-multirun is indeed everything we need:

Lemma 3.9 *Let* A *be a search automaton, over an alphabet* $\Sigma \times \{x, \bar{x}\}$. *Let* T *be a tree over* Σ *and let* T^v *be defined like in Lemma 3.1. Then the following two conditions are equivalent:*

(i) For every $v \in \{0,1\}^*$ *the automaton* A *accepts the tree* T^v.

(ii) There exists an accepting multirun Ψ *on* T *such that: (X) for every* $v \in \{0,1\}^*$ *there exist* q, q_0, q_1 *such that both* $\langle v0, q_0\rangle$ *and* $\langle v1, q_1\rangle$ *are in the domain of* Ψ, *that* $q \in co\Psi(v)$ *and that* $(q, \langle T(v), x\rangle) \to (q_0, q_1)$ *is a rule from* R.

Notice that by Lemma 3.6 and by the remark just above the last lemma we could equivalently write "exists an accepting uniform multirun" in the first line of the second item in the lemma.

Proof: (ii)⇒(i). Let Ψ be a multirun as in (ii) and let $v \in \{0,1\}^*$. We need to define an accepting initial run ρ of A on T^v. There are q, q_0, q_1 such that $q \in co\Psi(v)$, both $\langle v0, q_0\rangle$ and $\langle v1, q_1\rangle$ are in the domain of Ψ and $(q, \langle T(v), x\rangle) \to (q_0, q_1)$ is a rule from R. Define $\rho(v) = q$. For $w \geq v0$ define $\rho(w)$ as $\Psi_{v0,q_0}(w)$ and for $w \geq v1$ define $\rho(w)$ as $\Psi_{v1,q_1}(w)$. Now, if $v = \varepsilon$ then what we defined is already an initial accepting run. If not, let $v = y0$ for some y (the case when $v = y1$ is obviously symmetric). From the fact that $q \in co\Psi(v)$ and from the definition of $co\Psi$ we get that there are q_y and q_{y1} such that $\langle y1, q_{y1}\rangle \in dom(\Psi)$ and $(q_y, \langle T(y), \bar{x}\rangle) \to (q, q_{y1})$ is a rule of A. Define $\rho(y) = q_y$ and for $w \geq y1$ define $\rho(w) = \Psi_{y1,q_{y1}}(w)$. Then continue this process until the root of the tree is reached. It is not hard to check that the defined function ρ is indeed a run as needed.

(i)⇒(ii). For each v let ρ^v be an initial accepting run of A on T^v. We first define the domain of Ψ as the set of all pairs $\langle w, q\rangle$ such that there exist at least one v such that $\rho^v(w) = q$ and $v \not\geq w$. Then, for each $\langle w, q\rangle \in dom(\Psi)$ we fix one such v and define $\Psi_{w,q}(y) = \rho^v(y)$ for $y \geq w$. It is obvious that Ψ is an accepting multirun. To prove that Ψ satisfies condition (X) we need:

Lemma 3.10 $\rho^v(w) \in co\Psi(w)$ *for each* $v \in \{0,1\}^*$ *and each* $w \leq v$.

Proof: By item (i) of Definition 3.8 this is true for $w = \varepsilon$. Then use induction on the length of w. ■

Now, to finish the proof of Lemma 3.9 we notice that the triple $\rho^v(v), \rho^v(v0), \rho^v(v1)$ of states from Q is a correct candidate for the triple q, q_0, q_1 whose existence is postulated by condition (X): $\langle v0, \rho^v(v0)\rangle$ and $\langle v1, \rho^v(v1)\rangle$ are in the domain of Ψ, by the last lemma we have that $\rho^v(v) \in co\Psi(v)$ and, since ρ^v is a run of A on T^v, we have that $(\rho^v(v), \langle T(v), x\rangle) \rightarrow (\rho^v(v0), \rho^v(v1))$ is a rule from R. ■

3.2 An intermediate step

As an intermediate step between multiruns and automata we consider ranked multiruns:

Definition 3.11 *Let Ψ be a uniform multirun. Then* the rank on Ψ *is a function ϕ such that:*

(i) The domain of ϕ are all the tuples $\langle w, q, v\rangle$ such that $\langle w, q\rangle$ is in the domain of Ψ and $w \leq v$;

(ii) The values of ϕ are natural numbers from the set $\{1, 2, \ldots 2|Q|\}$;

(iii) $\Psi_{w_1,q_1}(v) = \Psi_{w_2,q_2}(v)$ if and only if $\phi(w_1, q_1, v) = \phi(w_2, q_2, v)$;

(iv) $\phi(w, q, v) \geq \phi(w, q, v0)$ and $\phi(w, q, v) \geq \phi(w, q, v1)$;

(v) If $\phi(w, q, v) = k$ and $\phi(w, q, v0) < k$ then there is no pair $\langle w', q'\rangle$ such that $\phi(w', q', v0) = k$. The same for $v1$: if $\phi(w, q, v) = k$ and $\phi(w, q, v1) < k$ then for no pair $\langle w', q'\rangle$ it can be that $\phi(w', q', v1) = k$.

Ranks are a way how we are going to organize the memory of $\forall A$ to store a uniform multirun: $\phi(w, q, v) = k$ can be understood as "in the node v the run $\Psi_{w,q}$ is kept in the register k". If two runs Ψ_{w_1,q_1} and Ψ_{w_2,q_2} are equal on some v then they remain equal forever (on the whole tree T_v) and we do not need to make a difference between them any more. This is why we rank them as equal on v, thus keeping them in the same memory register (item (iii)). It may also happen that two runs Ψ_{w_1,q_1} and Ψ_{w_2,q_2} were not equal on some v yet, but they are equal on $v0$ (or $v1$), and thus remain equal on T_{v0} (T_{v1}). Then, while moving from v to $v0$, we change the number of the register where one of the runs is remembered. Item (iv) gives us a hint how this will be done: we change the rank of the run which was ranked higher so far. But since we only can decrease the rank of a run, such a change can only happen (on a fixed path) finitely many times. This observation can be formalized as:

Lemma 3.12 *Suppose $w_1, w_2, \ldots$ is a path, the pair $\langle w, q\rangle$ is an element of the domain of a uniform multirun Ψ, $w \leq w_1$ and ϕ is a rank on Ψ. Then the sequence: $\phi(w, q, w_1), \phi(w, q, w_2), \phi(w, q, w_3) \ldots$ is non-increasing and thus constant from some point.*

Let us also explain the role of the last item in Definition 3.11. It is possible that some run Ψ_{w_1,q_1} has rank k on some v, and then, on $v' \geq v$ it already has rank $k' < k$. But the memory register k must be reused: there is another run Ψ_{w_2,q_2} which has rank k on v'. We need to give $\forall A$ a chance of seeing that Ψ_{w_1,q_1} and Ψ_{w_2,q_2}, despite being kept in the same memory register, are two different runs. The way we do it (in item (v)) is that we secure that there is a node v'' between v and v' when the register k is empty: no run has rank k on v''.

This subsection would be incomplete without:

Lemma 3.13 *For every uniform multirun Ψ of A there is a function ϕ which is a rank on Ψ.*

Proof: Use the same kind of inductive construction as in the proof of Lemma 3.5. The only new thing here is that we must show that it is enough to have only $2|Q|$ different ranks. But since two runs which are equal on v have the same rank on v we actually only need $|Q|$ different numbers to rank them. The remaining $|Q|$ are needed because of item (v) of the definition of rank. ■

3.3 The automaton $\forall A$

Now we are ready to define the automaton $\forall A$. The set $Q^\forall$ of the states of $\forall A$ consists of all possible tuples of the form: $\langle S, s_1, s_2, \ldots s_{2|Q|}\rangle$, where $S \subseteq Q$ and each s_i is either $\perp$ or is itself a tuple $\langle q, j_0, j_1\rangle$ where $q \in Q$ and j_0, j_1 are natural numbers from the set $\{1, 2, \ldots i\}$. This definition hardly comes as a surprise for a reader who understood the two previous subsections: the tuples s_i are the registers where $\forall A$ will remember the runs of some uniform multirun Ψ. If $s_i = \langle q, j_0, j_1\rangle$ in some node w then q is the value on w of all the runs with rank i, and j_0, j_1 are ranks of the runs on $w0$ and $w1$. Finally, S is where $co\Psi$ is going to be kept.

Having this explanation on mind it is easy to guess that:
$(\langle S, s_1, s_2, \ldots s_{2|Q|}\rangle, a) \rightarrow$
$(\langle S^0, s_1^0, s_2^0, \ldots s_{2|Q|}^0\rangle, \langle S^1, s_1^1, s_2^1, \ldots s_{2|Q|}^1\rangle)$
is a rule from the set $R^\forall$ of the rules of $\forall A$ if the following conditions hold:

(i) If $s_i = \langle q, j_0, j_1\rangle$ then neither $s_{j_0}^0$ nor $s_{j_1}^1$ is $\perp$. If $s_{j_0}^0 = \langle q_0, m_0, m_1\rangle$ and $s_{j_1}^1 = \langle q_1, n_0, n_1\rangle$ then $(q, \langle a, \bar{x}\rangle) \rightarrow (q_0, q_1)$ is a rule of the automaton A.

(ii) If $s_i = \langle q, j_0, j_1\rangle$ and $j_0 < i$ then $s_i^0 = \perp$. And also, if $s_i = \langle q, j_0, j_1\rangle$ and $j_1 < i$ then $s_i^1 = \perp$.

(iii) $q_1 \in S^1$ if and only if there is $q \in S$ and $s_i^0 = \langle q_0, j_0, j_1\rangle$ such that $(q, \langle a, \bar{x}\rangle) \rightarrow (q_0, q_1)$ is a rule of the original automaton A. Symmetrically, $q_0 \in S^0$ if and only if there is $q \in S$ and $s_i^1 = \langle q_1, j_0, j_1\rangle$ such that $(q, \langle a, \bar{x}\rangle) \rightarrow (q_0, q_1)$ is a rule of A.

(iv) There exist $q \in S$, $s_i^0 = \langle q_0, n_0, n_1\rangle$, and $s_{i'}^1 = \langle q_1, n_0', n_1'\rangle$ such that $(q, \langle a, x\rangle) \to (q_0, q_1)$ is a rule of A.

To end the construction of the automaton $\forall A$ we define $Q_0^\forall$ to be the set of such states $\langle S, s_1, s_2, \ldots s_{2|Q|}\rangle \in Q^\forall$ that $S = Q_0$ and for each accepting condition $F_i \in \{F_1, F_2 \ldots F_k\}$ of the automaton A we define $2|Q|$ accepting conditions $F_i^1, F_i^2 \ldots F_i^{2|Q|}$ of the automaton $\forall A$, where $\langle S, s_1, s_2, \ldots s_{2|Q|}\rangle \in F_i^j$ if and only if $s_j = \perp$ or $s_j = \langle q, j_0, j_1\rangle$ and $q \in F_i$.

It is easy to check that $\forall A$ satisfies the condition (L) and therefore is a search automaton: If $(\langle S, s_1, s_2, \ldots s_{2|Q|}\rangle, a) \to (\langle S^0, s_1^0, s_2^0, \ldots s_{2|Q|}^0\rangle, \langle S^1, s_1^1, s_2^1, \ldots s_{2|Q|}^1\rangle)$ is a rule and if i is such that s_i, s_i^0, and s_i^1 are not equal to $\perp$, then $s_i = (q, j_0, j_1)$, and, by item (ii), $i = j_0 = j_1$. By item (i), $s_i^0 = (q_0, n_0, n_1)$ and $s_i^1 = (q_1, m_0, m_1)$ where $(q, \langle a, \bar{x}\rangle) \to (q_0, q_1)$ is a rule of the automaton A. It follows that one of the states q, q_0, q_1 is in F_j and thus one of s_i, s_i^0, s_i^1 is in F_j^i.

Now, Lemma 3.1 will follow from Lemma 3.9 and from:

Lemma 3.14 *For every tree T over Σ the two conditions are equivalent:*

(i) There exists an initial and accepting run $\rho^\forall$ of $\forall A$ on T.

(ii) There exists an accepting multirun Ψ on T satisfying the condition (X) from Lemma 3.9

Proof: (i)⇒(ii). Let $\rho^\forall$ be an initial and accepting run of $\forall A$ on T. Let $co\Psi(w)$ be S where $\rho^\forall(w) = \langle S, s_1, s_2, \ldots s_{2|Q|}\rangle$. Define $dom(\Psi)$ as the set of all pairs $\langle w, q\rangle$ such that there exists i, j_0, j_1 such that $\rho^\forall(w) = \langle S, s_1, s_2, \ldots s_{2|Q|}\rangle$ and $s_i = \langle q, j_0, j_1\rangle$. We need to show how to extract the run $\Psi_{w,q}$ from $\rho^\forall$. First define $\Psi_{w,q}(w) = q$. Notice that if now $\rho^\forall(w0) = \langle S^0, s_1^0, s_2^0, \ldots s_{2|Q|}^0\rangle$, and $\rho^\forall(w1) = \langle S^1, s_1^1, s_2^1, \ldots s_{2|Q|}^1\rangle$ then (by item (i) of the definition of a rule from $R^\forall$) we have that $s_{j_0}^0 = \langle q_0, n_0, n_1\rangle$ and $s_{j_1}^1 = \langle q_1, m_0, m_1\rangle$ for some $q_0, q_1, n_0, n_1, m_0, m_1$. Thus we can define $\Psi_{w,q}(w0) = q_0$ and $\Psi_{w,q}(w1) = q_1$, and then, by induction, we can define in this manner $\Psi_{w,q}(v)$ for any $v \geq w$, so that $\Psi(w, q)$ is a run of A on $\bar{T}_w$. By definition of $Q_0^\forall$ and by item (iii) the three conditions of Definition 5 hold and, by item (iv), the multirun Ψ satisfies the condition (X). What still needs to be shown is that it is accepting. In order to prove it we fix w, q and show that $\Psi(w, q)$ is an accepting run of A on $\bar{T}_w$. Consider a path $x_1, x_2, x_3 \ldots$, where $x_1 \geq w$ and an accepting condition F_j from the set of accepting conditions of A. We want to show that elements from F_j occur infinitely many times in the sequence $\Psi_{w,q}(x_1), \Psi_{w,q}(x_2), \Psi_{w,q}(x_2), \ldots$.

But by the construction of $\Psi_{w,q}$ there exists, for each i, a number h_i such that $\rho^\forall(x_i) = \langle S^i, s_1^i, s_2^i, \ldots s_{2|Q|}^i\rangle$, and $s_{h_i}^i = \langle \Psi_{w,q}(x_i), n_0^i, n_1^i\rangle$ for some $n_0^i, n_1^i \leq h_i$. It also follows from the construction that h_{i+1} either equals to n_0^i or to n_1^i, so that $h_{i+1} \leq h_i$. The last observation implies that the sequence $h_1, h_2, h_3 \ldots$ stabilizes: there exist numbers i_0 and h such that $h_i = h$ if $i \geq i_0$. To finish the proof we consider the sequence $s_{h_{i_0}}^{i_0}, s_{h_{i_0+1}}^{i_0+1}, s_{h_{i_0+2}}^{i_0+2}, \ldots$, which is, as we said, the same as $s_h^{i_0}, s_h^{i_0+1}, s_h^{i_0+2}, \ldots$. None of the elements of the last sequence is $\perp$, so, since $\rho^\forall$ is accepting we get (by the accepting condition F_j^h) that infinitely many of the elements of the last sequence are of the form $\langle q', n, m\rangle$ where $q' \in F_j$.

(ii)⇒(i). Let Ψ be an accepting multirun on T satisfying the condition (X) from Lemma 3.9, and let ϕ be a rank on Ψ. Define $\rho^\forall(w)$ as a tuple $\langle S, s_1, s_2, \ldots s_{2|Q|}\rangle$ such that:

(i) $S = co\Psi(w)$

(ii) $s_i = \langle q, j_0, j_1\rangle$ if there exist $q' \in Q$ and $v \leq w$ such that $\Psi_{v,q'}(w) = q$ and $\phi(v, q', w) = i$ and also $\phi(v, q', w0) = j_0$ and $\phi(v, q', w1) = j_1$

(iii) $s_i = \perp$ if such v, q' as above do not exist.

It is easy to check that what we defined in this way is indeed an initial run of $A^\forall$. What still needs to be shown is that it is an accepting run. In order to prove it we consider a path $x_1, x_2, x_3 \ldots$ and an accepting condition F_k^l from the set of accepting conditions of $\forall A$. Let $\rho^\forall(x_i) = \langle S^i, s_1^i, s_2^i, \ldots s_{2|Q|}^i\rangle$. What we need to show is that the sequence $s_l^1, s_l^2, s_l^3 \ldots$ contains infinitely many $\perp$ symbols or it has infinitely many elements of the form $\langle q, n, m\rangle$ with $q \in F_k$. Suppose there are only finitely many $\perp$ symbols among the element of the sequence. So there is i_0 such that none of the s_l^i, where $i \geq i_0$, is a $\perp$. By the item (i) of the definition of rank this implies that there exist w, q such that $\phi(w, q, x_i) = l$ for $i \geq i_0$. Since $\Psi_{w,q}$ is an accepting run of A, we have that infinitely many of $\Psi_{w,q}(x_i)$ are in F_k. But if only $i \geq i_0$ then $s_l^i = \langle \Psi_{w,q}(x_i), n_i, m_i\rangle$ for some n_i, m_i. ■

References

[AF90] M.Ajtai, R.Fagin *Reachability is harder for directed than for undirected finite graphs*, Journal of Symbolic Logic, 55(1):113-150, 1990;

[AF97] S. Arora, R. Fagin *On winning strategies in Ehrenfeucht-Fraïssé games*, Theoretical Computer Science , 174:97-121, 1997;

[AFS98] M.Ajtai, R.Fagin, L.Stockmeyer *The Closure of Monadic $\mathcal{NP}$*, (extended abstract of [AFS00]) Proc. of 13th STOC, pp 309-318, 1998;

[AFS00] M.Ajtai, R.Fagin, L.Stockmeyer *The Closure of Monadic $\mathcal{NP}$*, Journal of Computer and System Sciences, vol. 60 (2000), pp. 660-716;

[A94] A. Arnold *An initial semantics for the μ–calculus on trees and Rabin's complementation lemma* Theoret. Comput. Science 148 (1995), 121–132.

[B60] J. Büchi, *On a decision method in restricted second order arithmetic*, in: Proc. of the International Congress on Logic, Methodology and Philosophy of Science 1960, Stanford University Press, Stanford (CA, USA), 1962, 1–11;

[EJ99] A. Emerson, C. Jutla *The complexity of tree automata and logics of programs* SIAM J. Comput. 29 (1999), 132–158.

[F74] R. Fagin *Generalized firsts-order spectra and polynomial time recognizable sets*, Complexity of computation, SIAM-AMS Proceedings, Vol 7 (R.M.Karp ed.) pp 43-73, 1974;

[F75] R. Fagin *Monadic Generalized spectra*, Zeitschrift fuer Mathematische Logik und Grundlagen der Mathematik, 21;89-96, 1975;

[GH82] Y. Gurevich, L. Harrington *Automata, trees and games*, in: Proc. 14th. Ann. ACM Symp. on the Theory of Computing (1982) 60–65;

[JM01] D. Janin, J. Marcinkowski *A Toolkit for First Order Extensions of Monadic Games*, Proceedings of STACS 2001, Springer LNCS 2010, pp 353-364

[L01] G. Lenzi *A new logical characterization of Büchi automata*, Proc. of STACS 2001, Springer LNCS 2010, pp 467-477;

[MT97] O. Matz, W. Thomas *The monadic quantifier alternation hierarchy over graphs is infinite*, Proc. 12th IEEE LICS 1997, pp 236-244;

[M63] D. Muller *Infinite sequences and finite machines*, in: Proc. 4th IEEE Symp. on Switching Circuit Theory and Logical Design (1963) 3–16;

[M99] J. Marcinkowski *Directed Reachability: From Ajtai-Fagin to Ehrenfeucht-Fraisse games*, Proceedings of CSL 99, Springer LNCS 1683, pp 338-349;

[MS87] D. Muller, P. Schupp *Alternating automata on infinite trees*, Theoret. Comput. Sci. 54 (1987), 267–276;

[R69] M. Rabin *Decidability of second–order theories and automata on infinite trees*, Trans. AMS 141 (1969), 1–35;

[R70] M. Rabin *Weakly decidable relations and special automata*, in: Y. Bar Hillel, ed., Mathematical Logic and Foundations of Set Theory, North–Holland, Amsterdam, 1970, 1–23;

[S94] T. Schwentick *Graph connectivity and monadic $\mathcal{NP}$*, Proc of 35th FOCS: 614-622,1994;

[S82] R. Streett *Propositional dynamic logic of looping and converse*, Inform. and Control 54 (1982), 121–141.

[Z98] W. Zielonka *Infinite games on finitely coloured graphs with applications to automata on infinite trees* Theoret. Comput. Sci. 200 (1998), 135–183.

4 Appendix: An elementary proof of the fact that EGFP is not in monadic Σ_1

We begin the proof with a definition:

Definition 4.1 *Let x be a vertex of a infinite binary tree T colored with some monadic relations $P_1, P_2, \ldots P_l$. Suppose k is some fixed natural number.*

(i) By the vertex type of x *we will mean the set $u(x) \subseteq \{1, 2 \ldots l\}$ such that $P_i(x)$ holds if and only if $i \in u(x)$.*

(ii) By the neighborhood type *of x we mean the triple*

$$U(x) = \langle u(x), u(x0), u(x1) \rangle$$

of the vertex types of x and its both children.

(iii) A tree type $\mathcal{U}$ is a function whose arguments are neighborhood types and values are natural numbers $0, 1, \ldots k$.

(iv) A tree T colored with $P_1, P_2, \ldots P_l$ has the tree type $\mathcal{U}(T)$ if for each neighborhood type U the number of vertices of this type in T is k-equal to $(\mathcal{U}(T))(U)$. Two numbers are understood to be k-equal if they are equal or if they are both greater or equal to k (in the last case one of them, or both, may be infinite).

(v) A perfect tree type is a pair $\langle \mathcal{U}, U \rangle$, where $\mathcal{U}$ is a tree type and U is a neighborhood type. A tree T has perfect tree type $\langle \mathcal{U}, U \rangle$ if $\mathcal{U}$ is its tree type and U is the neighborhood type of its root.

By locality of first order logic and by acyclicity of the infinite tree, in order to prove Theorem 2.4 it is enough to show:

Lemma 4.2 *For every natural number k there exists an infinite monadic tree T^0 colored with P_1 in a way satisfying property (*) such that for every T^1 being an extension of T^0 by monadic relations $P_2, P_3 \ldots P_l$ there is a monadic tree T^2 colored with the same relations as T^1, not satisfying property (*) and of the same perfect tree type as T^1*

Proof of Lemma 4.2 will occupy the rest of this section.

For two tree types $\mathcal{U}_1$ and $\mathcal{U}_2$ let $\mathcal{U}_1 \prec \mathcal{U}_2$ mean that $\mathcal{U}_1(U) \leq \mathcal{U}_2(U)$ for every neighborhood type U. Notice that for fixed l and k there is only some finite number of tree types. So the partial order $\prec$ is well-founded.

Definition 4.3 *Let T be the infinite monadic tree colored with monadic relations $P_1, P_2, P_3 \ldots P_l$, let T^1 be another tree of this kind and let x be a vertex of T. Then: $T[x \leftarrow T^1]$ is "T with T_x substituted with T^1" or, to be more precise, the infinite binary tree colored with monadic relations $P_1, P_2, P_3 \ldots P_l$ in such a way that $P_i(y)$ holds in $T[x \leftarrow T^1]$ if:*
(i) $P_i(y)$ holds in T and x is not a prefix of y or
(ii) $P_i(z)$ holds in T^1 and $y = xz$

It is easy to see that:

Lemma 4.4 *If x, y are two vertices of a colored tree T, such that $y < x$ then $\mathcal{U}(T_y) \prec \mathcal{U}(T_x)$.*

The last lemma and well-foundedness of $\prec$ give:

Lemma 4.5 *Let $x_1, x_2, x_3 \ldots$ be an infinite sequence of vertices of some T such that $x_i < x_{i+1}$ for each i. Then there is a number n_0 such that for every $n > n_0$ $\mathcal{U}(T_{x_{n_0}}) = \mathcal{U}(T_{x_n})$.*

Definition 4.6 *A tree type $\mathcal{U}$ will be called* ultimate *if for every neighborhood type U either $\mathcal{U}(U) = 0$ or $\mathcal{U}(U) = k$.*

Lemma 4.7 *Let $x_1, x_2, x_3 \ldots$ be an infinite sequence of vertices of some T such that $x_i < x_{i+1}$ for each i and let n_0 be the number from Lemma 4.5. Then the tree type $\mathcal{U}(T_{x_{n_0}})$ is ultimate.*

Lemma 4.8 *Let $x_1, x_2, x_3 \ldots$ be an infinite sequence of vertices of some T such that $x_i < x_{i+1}$ for each i and let n_0 be such a number that all T_{x_n} for $n > n_0$ have the same ultimate tree type $\mathcal{U}$ (such a number exists by Lemma 4.7). Then there exists $n_1 > n_0$ such that if $n > n_1$, if T^1 is a tree such that $\mathcal{U}(T^1) \prec \mathcal{U}$ and if the vertex type of the root of T^1 is the same as vertex type of x_n then the perfect type of T is equal to the perfect type of $T[x_n \leftarrow T^1]$.*

It is time now to define the tree T^0 from Lemma 4.2. In order to do it it is enough to specify the predicate P_1. Let m be the number of distinct vertex types. Then we put $P_1 = \{0^{mk} : k \in \mathcal{N}\}$. Obviously T^0 satisfies the property (*). Now consider some fixed tree T^1 being an extension of T^0 by monadic relations $P_2, P_3 \ldots P_l$. We need to show that there is a monadic tree T^2 colored with the same relations as T^1, not satisfying property (*) and of the same perfect tree type as T^1.

Let $x_i = 0^i$ and let n_1 be the constant from Lemma 4.8. We consider two numbers $j_0 < j_1$, both greater than n_1, and such that:
(i) the vertices 0^{j_0} and 0^{j_1} have the same vertex type in T^1
(ii) if $n = 0 \bmod m$ then either $n < j_0$ or $j_1 < n$.
Notice that the last condition implies that if x is on the path from 0^{j_0} to 0^{j_1} then $x \notin P_1$.

Now, define T^3 as the tree where $x \in P_i$ if and only if the following condition holds:
$x = 0^{r(j_1 - j_0)}y$ for some y, $0^{j_1 - j_0}$ is not a prefix of y, and $0^{j_0}y \in P_i$ holds in T^1.

Obviously the vertex type of the root of T^3 is the same as the vertex type of the root of $T^1_{j_0}$. It is also easy to see that $\mathcal{U}(T^3) \prec \mathcal{U}(T^1_{j_0})$. So by Lemma 4.8 the tree $T^2 = T^1[x_{j_0} \leftarrow T^3]$ has the same perfect type as T^1. To finish the proof of Lemma 2.4 we observe that since P_1 does not occur in T^3 there are only finitely many vertices in predicate P_1 in T^2 so the property (*) does not hold in T^2. ■

On Definability of Order in Logic with Choice

Taneli Huuskonen* Tapani Hyttinen*

Department of Mathematics
P.O.Box 4
FIN-00014 University of Helsinki
Finland
E-mail: {huuskone,thyttine}@helsinki.fi

Abstract

We will answer questions due to A. Blass and Y. Gurevich on definability of order in the first-order logic with Hilbert's epsilon operation. E.g., we will show that a linear ordering is almost surely definable in models with random choice.

There is a well-known discrepancy between computational and descriptive complexity in finite models. For instance, a finite automaton can check whether the number of elements in any given finite set is even or odd, even though this property is not expressible in either monadic Σ^1_1 or IFP (inflationary fixpoint logic). The difference apparently arises from the fact that the data in a computer's memory are always linearly ordered, even if the ordering is random. In the presence of a linear ordering, there is a much nicer correspondence between computational and descriptive complexity classes. In particular, parity becomes definable in both monadic Σ^1_1 and IFP. It is natural to ask if a similar descriptive strength can be obtained with weaker extensions of the various logics.

In recent years, several people have introduced strengthenings of the first-order logic by a choice operation in descriptive finite model theory, see the witness operation by S. Abiteboul and V. Vianu in [1] and Hilbert's epsilon operation, introduced by D. Hilbert and P. Bernays in § 8 of [4] in a restricted context, and discussed by A. Blass and Y. Gurevich in [2]. Choice operations are easy to define from a global linear ordering and hence easy to compute. Moreover, they are a natural concept in programming.

In this paper we study the expressive power of Hilbert's epsilon operation. In [2], the ε-logic is defined as follows. The syntax of the ε-logic is defined as that of the first-order logic with the following additional rule: If $\phi(v_i, \overline{y})$ is a formula of the ε-logic, then $\varepsilon v_i \phi(v_i, \overline{y})$ is a term. An ε-model $(\mathcal{A}, E)$ is a model $\mathcal{A}$ together with a choice operation E, i.e., E is a function from the power set of $\mathcal{A}$ to $\mathcal{A}$ such that for all non-empty $X \subseteq \mathcal{A}$, $E(X) \in X$. Then the interpretation of $\varepsilon v_i \phi(v_i, \overline{a})$ in an ε-model $(\mathcal{A}, E)$ is defined to be $E(\phi(\mathcal{A}, \overline{a}))$. Otherwise the semantics of the ε-logic is defined as the semantics of the first-order logic.

Very little is known about the expressive power of the ε-logic (in finite models). However, it is known that the ε-logic is more expressive than the first-order logic by the work of M. Otto ([6]). In [2], the following three questions were asked among others:

1. Is the standard order uniformly definable in ε-logic?
2. Is the last element of the standard order uniformly definable in ε-logic?
3. Is some linear ordering uniformly definable in ε-logic?

By the standard order we mean the usual ordering one gets from a choice-function: For finite ε-models $(\mathcal{A}, E)$ and $n < \omega$, we define $\mathcal{A}^n$ so that $\mathcal{A}^0 = \mathcal{A}$ and $\mathcal{A}^{n+1} = \mathcal{A}^n - \{E(\mathcal{A}^n)\}$. Then a is smaller or equal to b in the standard order if for all $n < \omega$, $a \in \mathcal{A}^n$ implies $b \in \mathcal{A}^n$. The existence of such an ordering shows that all ε-models are inherently rigid.

In this paper, we will give a negative answer to the first two questions (even) in finite ε-models. Notice that the standard order is easily definable in FO+ε+IFP, see [3], and notice further that this means that FO+ε+IFP is the same as PTIME.

The third question appears to be much harder, and our partial result stems from a failed attempt to solve it by a very straightforward random choice argument. Contrary to our expectations, we found out that there is an ε-formula which almost surely defines a linear ordering in finite ε-models. This leaves open an interesting question. Our result implies that any property which is almost surely definable on randomly ordered structures is also almost surely definable on ε-structures with a random choice. On the other hand, any property that is definable on all finite ε-structures is definable on all linearly ordered finite structures. We do not

*Research partially supported by the Academy of Finland, grant 40734, and the Mittag-Leffler Institute (both authors)

0-7695-1281-X/01 $10.00 © 2001 IEEE

know whether either of these implications can be reversed. The reversibility of the latter implication is, of course, the question we originally tried to answer.

1 Almost surely definable ordering

In this section we will sketch a proof of the fact that a linear ordering is almost surely definable in ε-logic.

We shall start the proof of the main theorem of this section (Theorem 1 below) by first explaining the general outline and looking at some of the details afterwards. In the more informal part, expressions such as "with high probability" mean that the limit probability of the claim holding in a random ε-model is 1. Our techniques resemble those that Matt Kaufmann used in [5] to handle monadic second-order logic, and his article gave us some ideas for simplifying ours.

We will use the following notion repeatedly. If $A \subseteq B$, $x \in B$ and $R \subseteq B^2$, we write $A_R[x]$ for the set $\{y \in A | R(x,y)\}$, and we say that x *R-codes* the set $A_R[x]$ in A.

Without loss of generality, we can assume that the vocabulary of our ε-models is empty. So, suppose we are given a random ε-model $\mathfrak{M}' = \langle M', E' \rangle$. We define a new random ε-model $\mathfrak{M} = \langle M, L^{\mathfrak{M}}; E \rangle$ inside $\mathfrak{M}'$, with a certain fixed vocabulary L that contains everything needed in the rest of the proof.

In the new model there is a random unary function F. With high probability, there is a point a whose preimage under F is somewhat smaller than $\log |M|$, but larger than $2 \log\log |M|$. Let A be the preimage of a. There is a subset $B \subseteq A$ of size logarithmic in $|A|$ such that a binary relation R is a linear ordering on B. Moreover, it is very likely that there is a parameter $b \in M$ that V_0-codes B in A, where V_0 is a random binary relation. Hence, we have a parameter definable set B with a definable linear ordering such that $|B| \geq \log\log\log m + 1$.

Every $x \in M$ V_1-codes a subset of B, where V_1 is another random binary relation. With the help of the choice operator, we can pick a set B_1 such that for each $C \subseteq B$ coded by some $x \in M$ there is exactly one $y \in B_1$ that V_1-codes C in B. On the other hand, the ordering of B induces, in a natural way, a linear ordering on B_1. With high probability, $|B_1| \geq \log\log m$. We can iterate this construction, this time looking at subsets of B_1 that are V_2-coded by elements of M. This way, we get definable sets B_2 and B_3, each of them with a definable linear ordering. Moreover, with high probability, $B_3 = M$; hence the whole model carries a definable linear ordering. Finally, we show how to get rid of the parameters that we used in the construction.

After this overview, we will state the claim and give the proof in more detail.

The formula

$$\forall xy(x = y \leftrightarrow (\varphi(x,y) \wedge \varphi(y,x))) \wedge \forall xy(\varphi(x,y) \vee \varphi(x,y))$$
$$\wedge \forall xyu((\varphi(x,y) \wedge \varphi(y,u)) \rightarrow \varphi(x,u)),$$

expressing the condition that $\varphi(x,y,\bar{z})$ defines a linear ordering, is denoted by $\mathrm{Lin}_\varphi(\bar{z})$, where $\bar{z}$ is a (possibly empty) sequence of parameters. For $n \in \mathbb{N}$, let $\mathfrak{S}_n$ be the set of all ε-models $(M; E)$ such that $M = \{0, \ldots, n-1\}$.

Theorem 1 *There is an ε-formula $\varphi(x,y)$ which defines a linear ordering in a random finite ε-model with limit probability* 1, *that is,*

$$\lim_{n\to\infty} \frac{|\{\mathfrak{M} \in \mathfrak{S}_n : \mathfrak{M} \models \mathrm{Lin}_\varphi\}|}{|\mathfrak{S}_n|} = 1.$$

Proof. The theorem will follow from the sequence of lemmas proved below. □

Assume that $\mathfrak{M}' = \langle M', E' \rangle$ is a finite ε-model with the empty vocabulary such that $|M'| = m + 9$ for some $m \geq 1$. We first define another model $\mathfrak{M} = \langle M, F, R, V_0, V_1, V_2, V_3; E \rangle$ within $\mathfrak{M}'$ such that $|M| = m$, F is a unary function, R is a tournament, i.e., an irreflexive binary relation such that exactly one of $R(x,y)$ and $R(y,x)$ holds, and the V_i, $i = 0,1,2,3$, are arbitrary binary relations. Moreover, if E' is chosen randomly, then F, V_0, V_1, V_2, V_3 and E are random and mutually independent. The tournament R, on the other hand, is directly defined from E, since we do not need to assume it to be random.

Firstly, define $a_0, \ldots, a_8$ to be the first nine elements in the standard linear ordering. That is, let $a_0 = E'(M')$, $a_1 = E'(M' \setminus \{a_0\})$, $a_2 = E'(M' \setminus \{a_0, a_1\})$, and so on. Then, let $M = M' \setminus \{a_0, \ldots, a_8\}$, and let $E = E' \restriction \mathcal{P}(M)$. Further, for $a \in M$, let $F'(a) = E'(\{a_0\} \cup M \setminus \{a\})$, and let

$$F(a) = \begin{cases} a, & \text{if } F'(a) = a_0, \\ F'(a), & \text{otherwise.} \end{cases}$$

Define R by $(x,y) \in R \Leftrightarrow E(\{x,y\}) \neq y$. (This implies, in particular, that R is irreflexive.) Finally, for $x, y \in M$, $i = 0,1,2,3$, define $V_i(x,y)$ to hold iff either $x \neq y$ and $E(\{x, y, a_{2i+1}, a_{2i+2}\}) \in \{x, a_{2i+1}\}$, or $x = y$ and $E(\{x, a_{2i+1}\}) = x$.

It is fairly easy to see that these definitions have the desired properties. For independence, it suffices to check that all of the defined relations depend on the values of E on different sets.

Since $\mathfrak{M}$ is definable inside $\mathfrak{M}'$ without parameters and so are all elements of $M' \setminus M$, it is clearly sufficient to define a linear ordering in $\mathfrak{M}$. So, from now on, we work in $\mathfrak{M}$. Throughout the rest of the section, $m = |M|$.

Lemma 2 *Let $F : M \to M$ be a random function. Then with probability approaching 1 as $m \to \infty$, there is $a \in M$ such that*

$$(\lfloor k_a/2 \rfloor)^{\lfloor k_a/2 \rfloor} \leq m \leq (2k_a)^{2k_a}$$

where $k_a = |F^{-1}[a]|$.

Proof. A much stronger result is known, but we sketch a simple argument sufficient to prove this weaker claim we need.

Firstly, the number of ways to choose a subset $X \subseteq M$ such that $|X| = k$ and an element $x \in M$ is $m\binom{m}{k}$. Each such pair (X, x) satisfies the condition $X \subseteq F^{-1}[x]$ with probability m^{-k}. Hence the probability that there exists $x \in M$ such that $|F^{-1}[x]| \geq k$ is at most

$$m^{1-k}\binom{m}{k} \leq m/k! \to 0,$$

if $\lfloor k/2 \rfloor^{\lfloor k/2 \rfloor} > m$. So, with limit probability 1, the first inequality is true for all $a \in M$.

On the other hand, let $M_0 \cup \ldots \cup M_{k-1}$, be a partition of M into k equal parts, supposing for simplicity that k divides m. Let $D_i = F[M_i]$. If there is some M_i such that $|D_i| \leq |M_i|/k = m/k^2$, then there must be some $a \in D_i$ such that $|F^{-1}[a] \cap M_i| \geq k$. If not, let $E_i = \cap_{j<i} D_i$. It can be shown by induction on i that $|E_i| \geq m/(3k/2)^{2i}$ with probability approaching 1. In particular, $E_{k-1} \neq \emptyset$. For $a \in E_{k-1}$, $|F^{-1}[a]| \geq k$.

The exact details of this proof are rather tedious and uninteresting, and we omit them. □

Lemma 3 *Let $n \in \mathbb{N}$, let A be a set such that $|A| \geq 2^n$, and let $R \subseteq A^2$ be a tournament. Then there is a set $B \subseteq A$ such that $|B| = n + 1$ and $R \restriction B$ is a strict linear order.*

Proof. Easy Ramsey-type induction on n. For $n = 0$, the claim is trivial. Assume then that the claim holds for $n = k$, and consider a set A such that $|A| \geq 2^{k+1}$. Choose an arbitrary element $a \in A$, and let $A_0 = \{x \in A : R(x, a)\}$ and $A_1 = \{x \in A : R(a, x)\}$. Now $A_0 \cup A_1 = A \setminus \{a\}$, and hence there is $i \in \{0, 1\}$ such that $|A_i| \geq 2^k$. By the induction hypothesis, there is a subset $B' \subseteq A_i$ such that $|B'| = k + 1$ and $R \restriction B'$ is a linear order. Now the set $B = B' \cup \{a\}$ witnesses the claim for $n = k + 1$. □

Lemma 4 *Let A be a subset of M such that $|A| = k$ with $\lfloor k/2 \rfloor^{\lfloor k/2 \rfloor} \leq m$, let $B \subseteq A$, and let $V_0 \subseteq M^2$ be a random binary relation. Then with probability approaching 1, there is $b \in M$ such that $B = A_{V_0}[b]$.*

Proof. A single element $y \in M$ fails to satisfy the condition with probability $1 - 2^{-k}$, independently of others. Hence the probability that no element satisfies the condition is

$$(1 - 2^{-k})^m \leq e^{-m/2^k} \to 0,$$

as $m \to \infty$. □

Corollary 5 *Let a be as in Lemma 2 and let $A = F^{-1}[a]$. Then with probability approaching 1, there are a parameter $b \in M$, a set $B \subseteq A$ and a linear ordering $<_B$ on B such that B and $<_B$ are ε-definable from the parameters a and b and that $m \leq (2^{|B|})^{2^{|B|}}$.*

Proof. Let $B \subseteq A$ be as in Lemma 3, define $x <_B y$ iff $R(x, y)$, for $x, y \in B$, and let $b \in M$ be such that $B = A_{V_0}[b]$. □

Lemma 6 *Let U, V be finite nonempty sets such that $|U| = u$, $|V| = v$, and let $f : U \to V$ be a random function.*

(i) The function f is one-to-one with probability at least $1 - u^2/v$.

(ii) The function f is onto with probability at least $1 - ve^{-u/v}$.

Proof. Easy. □

We will define sets B_i and linear orderings $<_i$ on them, respectively, for $i = 0, 1, 2, 3$, by recursion on i. Let $B_0 = B$, $<_0 = <_B$. Assume then that B_i and $<_i$ have been defined. Firstly, for $x \in M$, let $[x]_i = \{y \in M | (B_i)_{V_{i+1}}[y] = (B_i)_{V_{i+1}}[x]\}$, and let $B_{i+1} = \{x \in M | x = E([x]_i)\}$. Then, for $x, y \in B_{i+1}$, define $x <_{i+1} y$ iff there is some $z \in B_i$ such that $R_i(x, z)$ but not $R_i(y, z)$ and that for all $u <_i z$, we have $R_i(x, z) \leftrightarrow R_i(y, z)$.

Lemma 7 *The sets B_i and the relations $<_i$ are definable from the same parameters, a and b, as the set B and its ordering $<_B$ is. Moreover, each $<_i$ is a linear ordering on B_i.*

Proof. Easy induction on i.

Clearly, B_{i+1} is ε-definable from the same parameters as B_i. Moreover, $<_{i+1}$ is the partial ordering corresponding to the lexicographic ordering of $\mathcal{P}(B_i)$. Since B_{i+1} contains exactly one element from each equivalence class $[x]_i$, the ordering $<_{i+1}$ is actually linear. □

Lemma 8 *For $i = 0, 1, 2$, $|B_{i+1}| \geq \min(2^{|B_i|}, (\log m)^2)$ with limit probability* 1.

Proof. Consider the function $f_i : M \to \mathcal{P}(B_i)$, $f_i(x) = (B_i)_{V_{i+1}}[x]$. The set B_{i+1} contains exactly one element for each different value of f_i. If $2^{|B_i|} \leq \log^2 m$, then f_i is onto with limit probability 1, according to Lemma 6, and hence $|B_{i+1}| = 2^{|B_i|}$. Otherwise, it has at least $(\log m)^2$ different values. □

Corollary 9 *With limit probability 1, $|B_2| \geq (\log m)^2$.*

Proof. Since $m \leq (2^{|B_0|})^{2^{|B_0|}}$, we get $m \leq |B_1|^{|B_1|}$, and hence $|B_2| \log |B_2| \geq m$, which implies the claim for large enough m. □

Lemma 10 *With limit probability 1, $B_3 = M$.*

Proof. Let $f_2 : M \to \mathcal{P}(B_2)$ be as in the proof of Lemma 8. By Corollary 9 and Lemma 6, f_2 is one-to-one with limit probability 1. Hence it gets $|M|$ different values, and so $|B_3| = |M|$, thus $B_3 = M$. □

Now we have a linear order $<_3$ of the whole of M, but it is defined from two parameters. At this point, we can eliminate them with the epsilon operator.

Lemma 11 *Let $\xi(x, y, \bar{z})$ be an ε-formula and $\bar{a} \in M$ parameters such that $\xi(x, y, \bar{a})$ defines a linear ordering in $\mathfrak{M}$. Then there is a formula $\xi'(x, y)$ such that ξ' defines a linear ordering in $\mathfrak{M}$ without parameters.*

Proof. By induction on the number of parameters. If there are no parameters, there is nothing to prove. Let then $\xi(x, y, z_0, \ldots, z_k)$ be a formula with $k + 1$ parameters, $k \in \mathbb{N}$. Let $\psi(z_0, \ldots, z_k)$ be the formula asserting that ξ defines a linear ordering, and let $\xi_0(x, y, z_1, \ldots, z_k)$ be the formula

$$\xi(x, y, \varepsilon u(\psi(u, z_1, \ldots, z_k)), z_1, \ldots, z_k).$$

Now ξ_0 defines a linear ordering with k parameters, and therefore, by induction hypothesis, there is $\xi_0'(x, y)$ defining a linear ordering in $\mathfrak{M}$ without parameters. □

This lemma finishes the proof of Theorem 1.

2 Standard order is not definable in ε-logic

In this section we sketch a proof of a negative answer to the first question from [2]. We are forced to start proving everything from the definition of the ε-logic since there are no useful characterizations for the equivalence in the ε-logic. In fact, it seems very difficult to find e.g. a useful Ehrenfeucht-Fraïssé style characterization for equivalence in the ε-logic. Especially, this is the case if one restricts the equivalence to those sentences of the ε-logic which are independent from the choice of the choice operation (which is the most interesting fragment of the ε-logic).

The idea in the proof is simple (it will be tricky to find the right inductive hypotheses, though): We define two suitably different linear orderings $<$ and $<^*$ on a set A. Then we define a choice operation E on A so that $<$ will be the standard order but otherwise, whenever possible, E chooses $<^*$-least elements. Then, using the suitable difference between $<$ and $<^*$, we show that if a set is definable in (A, E) by a formula of the ε-logic, then it is essentially definable in $(A, <^*)$ by a first-order formula of roughly the same quantifier rank. Then we finish the proof by observing that next to nothing on $<$ is definable in $(A, <^*)$ by a first-order formula.

Above, in the phrase "essentially definable", the word "essentially" plays an important role. E.g. the second element in the canonical order is definable in (A, E) by a formula of the ε-logic but it will not be definable in $(A, <^*)$ by any first-order formula of reasonable quantifier rank.

In order to make the number of cases in the proofs small, we assume that all ε-formulas are in a form in which the quantifiers $\exists$ and $\forall$ do not appear. This is possible by the following observation.

Fact 12 *For every ε-formula $\phi(\bar{y})$ there is an ε-formula $\psi(\bar{y})$ such that the quantifiers $\exists$ and $\forall$ do not appear in ψ and for all ε-models $\mathcal{A}$ and sequences $\bar{a} \in \mathcal{A}$,*

$$\mathcal{A} \models \phi(\bar{a}) \Leftrightarrow \mathcal{A} \models \psi(\bar{a}).$$

By the quantifier rank $qr(\phi)$ of an ε-formula ϕ we mean the number of appearances of ε in ϕ (this definition, although unusual, will turn out to be convenient). We say that an ε-formula ϕ is ε-free if $qr(\phi) = 0$.

Let $N < \omega$ (and so $N = \{n < \omega | \ n < N\}$). By $\mathcal{A}_N = (A_N, E)$ we mean the following ε-model: $A_N = N \times N$. By $A_N^{\leq n}$ we mean the set of those $(a, b) \in A_N$ such that $b \leq n$. By $<$ we mean the lexicographic order of A_N, i.e., $(a, b) < (a', b')$ if $a < a'$ or $a = a'$ and $b < b'$. Notice that the pairs $(a, b) \in A_N$ may be considered as natural numbers $aN + b$, in which case $<$ is the usual ordering of the natural numbers. The ordering $<^*$ is defined as follows: $(a, b) <^* (a', b')$ if $b < b'$ or $b = b'$ and $a < a'$. Notice that if $x \in A_N^{\leq n}$ and $y \in A_N - A_N^{\leq n}$, then $x <^* y$. For $X \subseteq A_N$, we define $E(X)$ as follows: If for some $a < N - 1$ and $b < N$, $X = \{x \in A_N | \ x \geq (a, b)\}$, then $E(X) = (a, b)$ and otherwise $E(X)$ is the $<^*$-least member of X, if one exists, and $E(\emptyset) = (0, 0) (= \varepsilon(A_N))$. The subsets of A_N of the form $\{x \in A_N | \ x < (a, b)\}$, $a < N - 1$, are called standard (so, e.g., $\{x \in N \times N | \ x < (N - 1, 0)\}$ is not standard, this is important). Notice that $<$ is the standard order of $\mathcal{A}_N$. By $\mathcal{A}_N^*$ we mean the structure $(A_N, <^*)$.

By a $<^*$-formula we mean a first-order formula in the similarity type $\{<^*\}$ and the quantifier rank for such a formula is defined in the usual way. For $\bar{a}, \bar{b} \in A_N$, we write $(\mathcal{A}_N^*, \bar{a}) \equiv_n (\mathcal{A}_N^*, \bar{b})$ if $\bar{a}$ and $\bar{b}$ satisfy the same $<^*$-formulas up to quantifier rank n.

We will show that $<$ is not definable in $\mathcal{A}_N$ by an ε-formula of quantifier rank $< n$ assuming that N is large enough, say $N > 2^{4n+5}$. So all the time we assume that n and N are such that $N > 2^{4n+5}$. The following well-known fact is the reason for the choice of N (we state the fact in the form it will be used):

Fact 13 *Let $n < \omega$ and $\mathcal{A} = (A, <^*)$ be a linear ordering.*

(i) For $i < 4$, let $\bar{a}_i$ be a sequence of elements of A, for $i \in \{0, 2\}$, let a_i be the $<^*$-largest element of $\bar{a}_i$ and for $i \in \{1, 3\}$, let a_i be the $<^*$-smallest element of $\bar{a}_i$. Assume that $a_0 <^* a_1$, $a_2 <^* a_3$, in both intervals there

are at least $2^n - 1$ elements or the same number of elements, $(\mathcal{A}, \overline{a}_0) \equiv_n (\mathcal{A}, \overline{a}_2)$ and $(\mathcal{A}, \overline{a}_1) \equiv_n (\mathcal{A}, \overline{a}_3)$. Then $(\mathcal{A}, \overline{a}_0, \overline{a}_1) \equiv_n (\mathcal{A}, \overline{a}_2, \overline{a}_3)$.

(ii) Suppose $a, b \in A$ are such that there are $\geq 2^n - 1$ elements which are $<^*$-smaller than both a and b and $\geq 2^n - 1$ elements which are $<^*$-greater than both a and b. Then $(\mathcal{A}, a) \equiv_n (\mathcal{A}, b)$.

Definition 14 *(1) We say that a sequence $\overline{a} = (a_0, ..., a_p)$ of elements of A_N is (k,m)-good if for all $i \leq p$, $a_i \in A_N^{\leq m+1}$ implies $a_i \in A_N^{\leq k}$. Then we write $\overline{a}^0_{k,m}$ for $(a_i |\ a_i \in A_N^{\leq k})$ and $\overline{a}^1_{k,m}$ for $(a_i |\ a_i \in A_N - A_N^{\leq m+1})$. We say that $\overline{a}$ is (k,m,W)-good if $\overline{a}$ is (k,m)-good and $\overline{a}^1_{k,m} = (a_i |\ i \in W)$.*

(2) Let $k, m, p < N$. We say that (k,m)-good sequences $\overline{a}$ and $\overline{b}$ of elements of A_N are (k,m,p)-equivalent if the following holds:

(a) $(\mathcal{A}_N^*, \overline{a}) \equiv_p (\mathcal{A}_N^*, \overline{b})$,

(b) for all $i < lg(\overline{a})$, if $a_i \in A_N^{\leq k}$ or $b_i \in A_N^{\leq k}$, then $a_i = b_i$.

Instead of $\overline{a}^0_{k,m}$ and $\overline{a}^1_{k,m}$ we write usually just $\overline{a}^0$ and $\overline{a}^1$, k and m are always clear from the context.

We define $F : \mathbb{N} \to \mathbb{N}$ so that $F(0) = 3$ and $F(n+1) = 2F(n) + 1$ (i.e. $F(n) = 2^{n+2} - 1$).

Proposition 15 *For all $k, n < \omega$ and $N > 2^{4n+5}$, if $\overline{a}$ and $\overline{b}$ are finite sequences of A_N and they are $(k, k + F(n), 2n)$-equivalent, then $(A_N, \varepsilon, \overline{a})$ is equivalent in ε-logic to $(A_N, \varepsilon, \overline{b})$ up to quantifier rank n.*

For fixed $N < \omega$, we will prove the proposition by induction on n for those n for which $N > 2^{4n+5}$, and we will do this in a series of lemmas. However, in order to keep the induction going, we need to prove more. Let us repeat that from now on N is fixed.

To avoid notational difficulties we will give the following precise definition for $\mathcal{A} \models \phi(\overline{a})$, where $\overline{a} = (a_i)_{i < lg(\overline{a})}$ and $\phi(\overline{y})$ is either an ε-formula or a $<^*$-formula: We assume that all the variables in the formulas are from the set $\{v_i |\ i < \omega\}$ and $\mathcal{A} \models \phi(\overline{a})$ holds if ϕ is true in $\mathcal{A}$ when each free variable $v_i \in \overline{y}$ is interpreted as a_i. In addition, we assume that $\overline{y}$ always denotes a sequence of the form $(v_j)_{j<i}$.

In the following definitions we define formulas whose existence will be proved later.

Definition 16 *Let $\phi(v_i, \overline{y})$ be an ε-formula of quantifier rank r. For all $W \subseteq lg(\overline{y})$, k, $n \geq r$ and finite sequences $\overline{c}$ of elements of $A_N^{\leq k}$ we write $\psi^{\phi,c}_{k,n,W}(v_i, \overline{y})$ for a $<^*$-formula such that*

(1) $\psi^{\phi,\overline{c}}_{k,n,W}(v_i, \overline{y})$ is of quantifier rank $\leq 2r$,

(2) for all $(k, k + F(n), W)$-good $\overline{a} \in A_N$, if $\overline{a}^0 = \overline{c}$, then

$$\phi(A_N, \overline{a}) - A_N^{\leq k+F(n)-1} = \psi^{\phi,\overline{c}}_{k,n,W}(A_N, \overline{a}) - A_N^{\leq k+F(n)-1}.$$

Definition 17 *$\phi(v_i, \overline{y})$ be an ε-formula of quantifier rank r. For all $W \subseteq lg(\overline{y})$, k, $n \geq r$, $A \subseteq A_N^{\leq k+F(n)}$ and finite sequences $\overline{c}$ of elements of $A_N^{\leq k}$ we write $\theta^{\phi,A,\overline{c}}_{k,n,W}(\overline{y})$ for a $<^*$-formula such that*

(1) $\theta^{\phi,A,\overline{c}}_{k,n,W}(\overline{y})$ is of quantifier rank $\leq 2r$,

(2) for all $(k, k + F(n), W)$-good $\overline{a} \in A_N$, if $\overline{a}^0 = \overline{c}$, then $\mathcal{A}_N \models \theta^{\phi,A,\overline{c}}_{k,n,W}(\overline{a})$ iff $\phi(A_N, \overline{a}) \cap A_N^{\leq k+F(n)} = A$.

The following definition gives our induction assumption i.e. by induction on r we will show that every ε-formula of quantifier rank $\leq r$ is essentially equivalent to a $<^*$-formula.

Definition 18 *Let $\phi(v_i, \overline{y})$ be an ε-formula of quantifier rank r. We say that ϕ is essentially equivalent to a $<^*$-formula if for all k and $n \geq r$ such that $N > \max(k + F(n) + 3, 2^{4n+5})$, and for all $W \subseteq lg(y)$, $A \subseteq A_N^{\leq k+F(n)}$, and finite sequences $\overline{c}$ of elements of $A_N^{\leq k}$ the following holds:*

(1) $\psi^{\phi,\overline{c}}_{k,n,W}(v_i, \overline{y})$ exists,

(2) $\theta^{\phi,A,\overline{c}}_{k,n,W}(\overline{y})$ exists,

(3) if $\overline{a}$ is $(k, k + F(n), W)$-good, $C \subseteq A_N$ is standard and either $\phi(A_N, a) = C$ or $\neg\phi(A_N, a) = C$, then $C \subseteq A_N^{\leq k+F(n)}$.

If only (1) and (2) hold, then we say that $\phi(v_i, \overline{y})$ is weakly essentially equivalent to a $<^*$-formula.

Notice that if $C \subseteq A_N^{\leq k+F(n)}$ is standard, then $C \subseteq \{(a, b) \in A_N |\ a = 0,\ b \leq k + F(n)\}$ (assuming $N > k + F(n) + 1$).

The following lemma gives the means to handle the problem of standard sets. Notice that the empty set is standard.

Lemma 19 *Assume that $\phi(v_i, \overline{y})$ is an ε-formula of quantifier rank r.*

(i) If $\phi(v_i, \overline{y})$ is weakly essentially equivalent to a $<^*$-formula, then $\phi(v_i, \overline{y})$ is essentially equivalent to a $<^*$-formula.

(ii) Assume that $\phi(v_i, \overline{y})$ is essentially equivalent to a $<^*$-formula. Let $n > r$ and k be such that $N > \max(k + F(n) + 3, 2^{4n+5})$. Suppose $\overline{a}$ and $\overline{b}$ are $(k, k + F(n), W)$-good, $(k, k + F(n), 2r + 1)$-equivalent, $\overline{a}^0 = \overline{b}^0 = \overline{c}$ and $\phi(A_N, \overline{a})$ is standard. Then $\phi(A_N, \overline{a}) = \phi(A_N, \overline{b})$.

Proof. (i): Let k, $n \geq r$ and $W \subseteq lg(\overline{y})$ be as in Definition 18 and let $\overline{a}$ be $(k, k + F(n), W)$-good. By Fact 13, for all $b, c \in A_N^{\leq k+F(n)} - A_N^{\leq k+F(n)-1}$, $(\mathcal{A}_N^*, b, \overline{a}) \equiv_{2r} (\mathcal{A}_N^*, c, \overline{a})$. So by Definition 18 (1),

(*) $\mathcal{A}_N \models \phi(b, \overline{a}) \leftrightarrow \phi(c, \overline{a})$.

But by the definition of a standard set C, $C \cap (A_N^{\leq k+F(n)} - A_N^{\leq k+F(n)-1}) \neq A_N^{\leq k+F(n)} - A_N^{\leq k+F(n)-1}$ and if $C \cap$

$(A_N^{\leq k+F(n)} - A_N^{\leq k+F(n)-1}) = \emptyset$, then $C \subseteq A_N^{\leq k+F(n)-1}$. With (*), this implies the claim.

(ii): We show that $\phi(A_N, \bar{b}) - A_N^{\leq k+F(n)} = \emptyset$, the rest is easy. Assume not. Let d witness this. Since $\bar{a}$ is $(k, k + F(n))$-good and $(\mathcal{A}_N^*, \bar{a}) \equiv_{2r+1} (\mathcal{A}_N^*, \bar{b})$, we can choose, by Fact 13 (i), $d' \in A_N - A_N^{\leq k+F(n)}$ so that $\bar{a}^\frown(d')$ and $\bar{b}^\frown(d)$ are $(k, k+F(n)-1, 2r)$-equivalent. But then $\mathcal{A}_N^* \models \psi_{k,n,W}^{\phi,\bar{c}}(d', \bar{a})$, a contradiction. □

We skip the proof of the following lemma.

Lemma 20 *(i) If ϕ is an ε-free atomic formula, then it is essentially equivalent to a $<^*$-formula.*

(ii) If ε-formula ϕ is essentially equivalent to a $<^*$-formula, then so is $\neg\phi$.

(iii) If ε-formulas $\phi(v_i, \bar{y})$ and $\phi'(v_i, \bar{y})$ are essentially equivalent to a $<^*$-formula, then so is $\phi \wedge \phi'$. □

Lemma 21 *If an ε-formula $\phi(v_i, v_j, \bar{y})$, $\bar{y} = (y_l)_{l<lg(\bar{y})}$, is essentially equivalent to a $<^*$-formula, then so is $\phi^* = (\varepsilon v_i \phi(v_i, v_j, \bar{y}) = z)$, where $z = v_j$ or y_l for some $l < lg(\bar{y})$.*

Proof. Without loss of generality we may assume that $j = lg(\bar{y})$ and $i = j + 1$. Let ϕ be of quantifier rank p. We assume that $z = v_j$, the other case is similar (and easier). Let $k, n, m, W, \bar{c}$ and $A \subseteq A_N^{\leq k+F(n)}$ be as in Definition 18 for $r = p + 1$. Assume that $\bar{a}$ and $\bar{b}$ are $(k, k + F(n), 2r)$-equivalent, and $\bar{a}^0 = \bar{b}^0 = \bar{c}$. If $c \in A_N^{\leq k+F(n)}$ is such that $\varepsilon v_i \phi(v_i, c, \bar{a}) = c$ holds then by the induction assumption and Lemma 19, also $\varepsilon v_i \phi(v_i, c, \bar{b}) = c$ holds (sequences $\bar{a}^\frown(c)$ and $\bar{b}^\frown(c)$ are always either $(k, k + F(n-1), W \cup \{j\})$-good or $(k + 1 + F(n-1), k + 1 + 2F(n-1), W)$-good). With this one can see that $\theta_{k,n,W}^{\phi^*,A,\bar{c}}(\bar{y})$ exists, i.e. (2) in Definition 18 holds.

For item (1) in Definition 18, we notice that by the induction assumption, if $c \notin A_N^{\leq k+F(n)-1}$, $\bar{a}$ is $(k, k + F(n))$-good and $\neg\phi(A_N, c, \bar{a})$ is standard or $\phi(A_N, c, \bar{a})$ is empty, then $\varepsilon v_i \phi(v_i, c, \bar{a}) \neq c$. Then one can check that

$$\theta_{k+1,n-1,W\cup\{j\}}^{\phi,\emptyset,\bar{c}}(v_j, \bar{y}) \wedge$$

$$\exists v_i(\psi_{k,n-1,W\cup\{j\}}^{\phi,\bar{c}}(v_i, v_j, \bar{y}) \wedge v_i = v_j) \wedge$$

$$\forall v_i(\psi_{k,n-1,W\cup\{j\}}^{\phi,\bar{c}}(v_i, v_j, \bar{y}) \rightarrow (\neg(v_i <^* v_j) \vee \eta(v_i, \bar{c})))$$

is as wanted, where $\eta(v_i, \bar{c})$ says that for some $l < lg(\bar{c})$, there are less than $2^{2p} - 1$ elements x such that $c_l <^* x <^* v_i$ and if $\bar{c}$ is the empty sequence, then $\eta(v_i)$ says that there are less than $2^{2p} - 1$ elements $< v_i$. □

The following lemma can be proved using ideas from the proof of Lemma 21.

Lemma 22 *If ε-formulas $\phi(v_i, v_j, \bar{y})$ and $\phi'(v_i, v_j, \bar{y})$ are essentially equivalent to $<^*$-formulas, then so is $\phi^* = (\varepsilon v_i \phi(v_i, v_j, \bar{y}) = \varepsilon v_i \phi'(v_i, v_j, \bar{y}))$.* □

Proof of Proposition 15. Follows from Lemmas 20, 21 and 22. □

Conclusion 23 *The standard order is not uniformly definable in ε-logic.*

Proof. For a contradiction, assume that the standard order is definable by an ε-formula of quantifier rank n. Let $N < \omega$ be such that $N > 2^{4n+5}$. Then by Fact 13, it is easy to see that $((N-2, N-3), (N-3, N-2))$ is $(1, 1 + F(n), 2n)$-equivalent to $((N-2, N-3), (N-2, N-2))$ but $(N-2, N-3) > (N-3, N-2)$ and $(N-2, N-3) < (N-2, N-2)$. By Proposition 15, we have a contradiction. □

Conclusion 24 *The last element in the standard order is not uniformly definable in ε-logic.*

Proof. We define an ordering $<^+$ to A_N as follows: $x <^+ y$, if either $x <^* y$ and $x \in A_N^{\leq N-2}$ or $x, y \in A_N - A_N^{\leq N-2}$ and $y <^* x$. Also a new choice operation E^+ is defined. This is defined exactly as E using $<^+$ in place of $<^*$. Then, as above, we can see that if $a, b \in A_N - A_N^{\leq 2+F(n)}$, $N > 2^{4n+5}$, are such that $(A_N, <^+, a) \equiv_{2n} (A_N, <^+, b)$, then (A_N, E^+, a) is equivalent in ε-logic to (A_N, E^+, b) up to quantifier rank n. This implies the claim (the last element in the standard order is the $<^+$-first element of $A_N - A_N^{\leq N-2}$). □

References

[1] S. Abiteboul and V. Vianu, Nondeterminism in logic-based languages, Annals of Mathematics and Artificial Intelligence, vol. 3, 151–186.

[2] A. Blass and Y. Gurevich, The logic of choice, Journal of Symbolic Logic, vol. 65, 1264–1310, 2000.

[3] H.-D. Ebbinghaus and J. Flum, Finite Model Theory, Springer, 1995.

[4] D. Hilbert and P. Bernays, Grundlagen der Mathematik I, Springer, 1934.

[5] M. Kaufmann, A counterexample to the 0-1 law for existential monadic second-order logic, manuscript.

[6] M. Otto, Epsilon-logic is more expressive than first-order logic over finite structures, Journal of Symbolic Logic, vol. 65, 1749–1757, 2000.

Invited Talk

Session 5

A second-order system for polytime reasoning using Grädel's theorem*

Stephen Cook and Antonina Kolokolova
University of Toronto
{sacook,kol}@cs.toronto.edu

Abstract

We introduce a second-order system V_1-Horn of bounded arithmetic formalizing polynomial-time reasoning, based on Grädel's [15] second-order Horn characterization of P. *Our system has comprehension over* P *predicates (defined by Grädel's second-order Horn formulas), and only finitely many function symbols. Other systems of polynomial-time reasoning either allow induction on* NP *predicates (such as Buss's S_2^1 or the second-order V_1^1), and hence are more powerful than our system (assuming the polynomial hierarchy does not collapse), or use Cobham's theorem to introduce function symbols for all polynomial-time functions (such as Cook's PV and Zambella's* P*-def). We prove that our system is equivalent to QPV and Zambella's* P*-def. Using our techniques, we also show that V_1-Horn is finitely axiomatizable, and, as a corollary, that the class of $\forall\Sigma_1^b$ consequences of S_2^1 is finitely axiomatizable as well, thus answering an open question.*

1 Introduction

1.1 Bounded Arithmetic

Here *Bounded Arithmetic* loosely refers to a collection of weak formal theories of arithmetic connected to the complexity classes P (polynomial time) and PH (the polynomial hierarchy) (see [3, 17, 20, 6, 2]). Study of these theories is motivated partly by the fundamental questions in complexity theory: Does P $\neq$ NP? Does PH collapse? An early example is the equational theory PV (for "Polynomially Verifiable") [8], which includes function symbols for all polynomial-time functions, defining equations for them based on Cobham's theorem, and a proof rule implementing induction on binary notation. The idea is that an equation is provable in PV iff it can be uniformly verified using only polytime concepts.

Later Buss [3] introduced a hierarchy of first-order theories $\langle S_2^1, S_2^2, S_2^3 ...\rangle$ corresponding to the levels of the polynomial hierarchy. In particular S_2^1 corresponds to polynomial time, in the sense that a function $f : \mathbb{N} \to \mathbb{N}$ is polynomial-time computable iff there is a so-called Σ_1^b formula $A(x, y)$ defining the graph of f such that $S_2^1 \vdash \forall x \exists y A(x, y)$. Here Σ_1^b formulas are certain bounded formulas which semantically represent precisely the NP predicates. S_2^1 includes PIND (induction on notation) axioms for all Σ_1^b formulas.

We define QPV (*quantified PV*) to be the first-order theory with the same language as the equational theory PV, and whose axioms are the theorems of PV. Buss [3] proves that every $\forall\Sigma_1^b$ theorem of S_2^1 is a theorem of QPV. However [21] proves that the induction axioms for S_2^1 (which are not $\forall\Sigma_1^b$ formulas) are not all theorems of QPV, unless PH collapses. (Complexity theorists generally assume that PH does not collapse.) The theory V_1-Horn that we introduce in this paper turns out to be equivalent of QPV (rather than S_2^1), as explained below.

An important open question is whether the union theory S_2 of Buss's hierarchy $\langle S_2^i \rangle$ of theories is finitely axiomatizable. As shown in [21, 5, 31], this happens iff S_2 proves that PH collapses. Since each of the theories S_2^i is finitely axiomatizable, it is immediate that S_2 is finitely axiomatizable iff the hierarchy $\langle S_2^i \rangle$ collapses. Thus the hierarchy $\langle S_2^i \rangle$ collapses iff S_2 proves that PH collapses.

The theory S_2^1 is finitely axiomatizable because it has a finite language, and its infinite induction scheme for Σ_1^b formulas follows from finitely many induction axioms, including one for a formula representing an NP-complete predicate. It does not make sense to ask whether QPV is finitely axiomatizable, because it has infinitely many function symbols. However [3] shows that PV is equivalent to the Σ_1^b consequences of S_2^1, so it makes sense to ask whether the latter are finitely axiomatizable. We answer this affirmatively in this paper by showing that our theory V_1-Horn is finitely axiomatizable (essentially by Σ_1^b formulas) and is equivalent to QPV.

Buss [3] introduced two hierarchies of so-called second-order theories, including a theory for polynomial space and

*An expanded version of this paper is available as ECCC report number TR01-024 [7].

0-7695-1281-X/01 $10.00 © 2001 IEEE

one for exponential time. (All "second-order" theories that we discuss are actually two-sorted first-order theories; with one sort for numbers and the other for finite bit strings.) Razborov [26] argues at length that a related second-order theory called V_1^1 can nicely formalize existing lower bound proofs on the complexity of explicitly given Boolean functions, and points out that by the "RSUV isomorphism" [25, 29], V_1^1 is equivalent to the first-order theory S_2^1 and hence captures polynomial-time reasoning.

Zambella [31] introduced an elegant presentation for second-order theories such as V_1^1, and we use this style here to present our theory V_1-Horn. One of Zambella's second-order theories, P-def, has function symbols for all polynomial-time functions, and can be shown to be equivalent to the first-order theory QPV by the RSUV isomorphism. We show that V_1-Horn is equivalent to P-def, in the sense that every theorem of V_1-Horn is a theorem of P-def, and every theorem of P-def can be translated into a theorem of V_1-Horn by replacing function symbols by their definitions in V_1-Horn.

1.2 Descriptive Complexity

The first connection between finite model theory and complexity theory goes back to Fagin's 1974 result [13] showing that a language is in NP iff it corresponds to the set of finite models of an existential second-order formula. Later Stockmeyer [28] extended this result, characterizing the polynomial hierarchy as the class of sets of finite models of all second-order formulas.

Finding an elegant descriptive-style characterization of P proved more illusive. One such characterization of P uses the first-order logic augmented with the successor relation and the least fixed-point operator [30, 18]. Later Leivant [22, 23] found a second-order characterization of P using the notion of "controlled computational formula", which is related to Horn formula. (The motivation for using Horn formulas comes from the existence of a simple polynomial-time algorithm for solving the satisfiability problem for propositional Horn formulas.) Finally Grädel [14, 15] found an elegant descriptive characterization of P using $SO\exists$-Horn (second-order existential Horn) formulas with successor.

1.3 Outline

In Section 2 we give the syntax and intended semantics of second-order formulas and show that certain syntactic classes of formulas represent the relations in certain corresponding complexity classes. In particular, the Σ_1^B-Horn (second-order existential Horn) formulas represent the polynomial-time predicates (by Grädel's theorem). We define various second-order theories in Section 3, including our theory V_1-Horn and the theory V^0 corresponding to the complexity class AC^0. The theory V_1-Horn uses a comprehension axiom scheme for the Σ_1^B-Horn formulas. In Section 4 we show that V_1-Horn proves the equivalence of each formula in several broad syntactic classes to a Σ_1^B-Horn formula. Section 5 contains the description of the main tool needed for later sections, namely representing the Horn satisfiability algorithm in V_1-Horn by a Σ_1^B-Horn formula and proving its correctness in V_1-Horn. In Section 6 we construct a conservative extension V_1-Horn(FP) of V_1-Horn by introducing function symbols for polynomial-time functions, and show the equivalence of this and Zambella's P-def[31]. Finally, in Section 7 we demonstrate that both V^0 and V_1-Horn are finitely axiomatizable, and show that this implies that the $\forall\Sigma_1^b$ consequences of S_2^1 are finitely axiomatizable.

2 Second-order formulas and complexity classes

The prototype for the underlying language of V_1-Horn is the language of second-order bounded arithmetic introduced by Buss [3]. However our language is closer to the nicer second-order language introduced by Zambella [31], in that we eliminate the superscript terms t tagging second-order variables X^t and instead introduce a bounding function $|X|$.

Our language $\mathcal{L}_A^2$ has two sorts, called first-order and second-order. (The intention is that first-order objects are natural numbers and second-order objects are finite sets of natural numbers, or finite binary strings.) First-order variables are denoted by lower case letters $a, b, i, j, ..., x, y, z$, and second-order variables are denoted by upper-case letters $P, Q, ..., X, Y, Z$.

The first-order function and predicate symbols of $\mathcal{L}_A^2$ are the standard symbols $\{0, 1, +, \cdot, \leq, =\}$ of Peano Arithmetic. To these we add the unary length function symbol $|\ |$, which takes second-order objects to first-order objects, and the binary membership predicate symbol $\in$.

For every second-order variable X we form a first-order term $|X|$ called a *length term*. The first-order terms of $\mathcal{L}_A^2$ are built from 0, 1, first-order variables, and length terms using the function symbols + and $\cdot$. The only second-order terms are second-order variables.

The atomic formulas of $\mathcal{L}_A^2$ have one of the forms $s = t, s \leq t, t \in X$, where s and t are first-order terms and X is a second-order variable. We usually write $X(t)$ instead of $t \in X$. Formulas are built from atomic formulas using the propositional connectives $\wedge, \vee, \neg$, the first-order quantifiers $\forall x, \exists x$ and the second-order quantifiers $\forall X, \exists X$.

We use the usual abbreviations $s \neq t$ for $\neg s = t$ and $s < t$ for $s \leq t \wedge s \neq t$. Bounded first-order quantifiers get their usual meaning: $\forall x \leq t \phi$ stands for $\forall x(x \leq t \rightarrow \phi)$

and $\exists x \leq t\phi$ stands for $\exists x(x \leq t \wedge \phi)$. We also use bounded second order quantifiers: $\forall X \leq t\phi$ stands for $\forall X(|X| \leq t \rightarrow \phi)$ and $\exists X \leq t\phi$ stands for $\exists X(|X| \leq t \wedge \phi)$.

In the standard model for $\mathcal{L}_A^2$ first-order variables range over $\mathbb{N}$, and second-order variables range over finite subsets of $\mathbb{N}$. If X is the empty set, then $|X|$ is interpreted as 0, otherwise $|X|$ is interpreted as one more than the largest element of the finite set X. The symbols 0,1,+ $\cdot$, $\in$ get there usual interpretations.

In complexity theory a member of a language is often taken to be a binary string, but from our "second-order" point of view we take it to be a finite subset X of $\mathbb{N}$. To relate this to the string point of view we code a finite set X by the binary string X', where X' is the empty string if X is the empty set, and otherwise X' is the binary string $x_0x_1, ..., x_{n-1}$ of length $n = |X|$ such that $x_i = 1 \iff i \in X, 0 \leq i \leq n-1$. (Thus all nonempty string codes end in 1.) If L is a set of finite subsets of $\mathbb{N}$, then the corresponding set of strings is $L' = \{X' \mid X \in L\}$. If C is a standard complexity class such as $\mathtt{AC}^0$, P or NP, then our second-order reinterpretation of C is $\{L \mid L' \in \mathtt{C}\}$. Since the complexity classes considered here are robust, this reinterpretation will come out the same for any reasonable string coding method.

The role of first-order objects in our theories is that of members of second-order objects, or equivalently as position indices for binary strings. Thus in determining the complexity of a set of natural numbers we code a natural number i using unary notation; that is as a string i' of 1's of length i.

Definition 2.1. If $\phi(\bar{z}, \bar{Y})$ is a formula of $\mathcal{L}_A^2$ whose free variables are among $z_1, ..., z_k, Y_1, ..., Y_\ell$ then ϕ *represents* a $k + \ell$-ary relation R^ϕ as follows. If $a_1, ..., a_k$ are natural numbers and $B_1, ..., B_\ell$ are finite sets of natural numbers, then $\langle a_1, ..., a_k, B_1, ..., B_\ell \rangle$ satisfies R^ϕ iff $\phi(a_1, ..., a_k, B_1, ..., B_\ell)$ is true in the standard model.

If C is a complexity class, then we make sense of the statement "R^ϕ is in C" using the string encodings described above. In particular, a relation $R(x_1, ..., x_k, Y_1, ..., Y_m)$ is in P iff it is recognizable in time bounded by a polynomial in $(x_1, ..., x_k, |Y_1|, ..., |Y_m|)$.

We now define the classes Σ_i^B and Π_i^B of bounded second-order formulas. (A formula is *bounded* if all its quantifiers are bounded.) Σ_0^B and Π_0^B both denote the class of bounded formulas with no second-order quantifiers. We define inductively Σ_{i+1}^B as the least class of formulas containing Π_i^B and closed under disjunction, conjunction, and bounded existential second-order quantification . The class Π_{i+1}^B is defined dually.

The classes Σ_i^B and Π_i^B are the formulas in our (Zambella's) simplified language $\mathcal{L}_A^2$ which correspond to the classes $\Sigma_i^{1,b}$ and $\Pi_i^{1,b}$ in Buss's prototype second-order language [3, 20]. They are the second-order analogs of the first-order formula classes Σ_i^b and Π_i^b, where sharply-bounded quantifiers correspond to our bounded first-order quantifiers.

The formulas Σ_1^B represent precisely the NP relations, and more generally for $i > 1$ the Σ_i^B formulas represent the Σ_i^p relations in the polynomial hierarchy and Π_i^B represent the Π_i^p relations [3, 20]. The formulas Σ_0^B represent precisely the uniform $\mathtt{AC}^0$ relations, which are the same as the class FO (First Order) of descriptive complexity [1] (see Chapter 1 of [19]).

We now define the formulas corresponding to polynomial time. Recall that a CNF (conjunctive normal form) formula is a conjunction of clauses of the form $(L_1 \vee ... \vee L_m), m \geq 1$ where each L_i is a *literal*; that is an atomic formula or a negated atomic formula.

Definition 2.2. A formula ϕ of $\mathcal{L}_A^2$ is *Horn with respect to the second-order variables* $P_1, ..., P_k$ if ϕ is quantifier-free in CNF and in every clause there is at most one positive literal of the form $P_i(t)$ (called the *head* of the clause) and no terms of the form $|P_i|$. (We do allow length terms $|X|$ and any number of positive literals $X(t)$, where X is not among $\{P_1, ..., P_k\}$.) A formula is Σ_1^B-Horn if it has the form

$$\exists P_1 ... \exists P_k \forall x_1 \leq t_1 ... \forall x_m \leq t_m \phi \qquad (1)$$

where $k, m \geq 0$ and ϕ is Horn with respect to $P_1, ..., P_k$, and the bounding terms t_i do not involve $x_1, ..., x_m$. More generally a formula is Σ^B-*Horn* if it has the above form except that each second-order quantifier can be either $\exists$ or $\forall$. A formula is Π_1^b *Horn with respect to* $P_1, ..., P_k$ if it has the form (1) with the existential quantifiers omitted.

Notice that our definition of Σ_1^B-Horn is somewhat different from Grädel's original definition of second-order existential Horn formula, as explained before Theorem 2.3. Also note that the second-order quantifiers in Σ_1^B-Horn and Σ^B-Horn formulas are not bounded. However, since no occurrence of $|P_i|$ is allowed, each such formula is equivalent in the standard model to one in which every quantifier $\exists P_i$ or $\forall P_i$ is bounded by a term t which is an upper bound on all terms u such that $P_i(u)$ occurs in the formula. On the other hand, if occurrences of $|P_i|$ were allowed, then an unbounded quantifier $\exists P_i$ can code an unbounded number quantifier $\exists |P_i|$ and hence undecidable relations would be representable.

It is often convenient to treat second-order objects as multi-dimensional arrays, instead of one-dimensional strings or sets. An easy way to do so is to use a pairing function $< \cdot, \cdot >$, defined by

$$< x, y >= (x + y)(x + y + 1) + 2y \qquad (2)$$

This function is a one-one map from $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$, and it is represented by a term in our language. It is easily general-

ized to k-tuples by defining $\langle x_1, ..., x_k \rangle$ by the recursion

$$\langle x \rangle = x, \qquad \langle x_1, ..., x_{k+1} \rangle = \langle \langle x_1, ..., x_k \rangle, x_{k+1} \rangle \quad (3)$$

Thus, any finite set P can be treated as a set of k-tuples of variables; $P(x_1, ..., x_k)$ is defined to be $P(< x_1, ..., x_k >)$.

The theorem below is similar to part of Grädel's Theorem 5.2 [14] (see also Chapter 7 of [27]), which is stated in the context of descriptive complexity theory. There are technical differences: Grädel's language is more general in that it allows predicate symbols of arbitrary arity, but these can be simulated by the pairing function as just explained. On the other hand our language is more general in that it allows interpreted function symbols + and · and terms $|Y_i|$, as well as universally quantified number variables whose range goes up to any polynomial in the size of the inputs. However none of these generalizations takes us outside the polynomial-time relations.

Theorem 2.3. *A relation $R(z_1, ..., z_k, Y_1, ..., Y_m)$ is in* P *iff it is representable by a Σ_1^B-Horn formula Ψ. Further Ψ can be chosen with only one existentially quantified second-order variable, and only two universally quantified first-order variables.*

Example. (**PARITY**(X)) This is a Σ_1^B-Horn formula which is true for strings X that contain an odd number of 1's. It encodes a dynamic-programming algorithm for computing parity of X: $P_{odd}(i)$ is true (and $P_{even}(i)$ is false) iff the prefix of X of length i contains an odd number of 1's.

$$\begin{aligned}
&\exists P_{even} \exists P_{odd} \forall i < |X| \\
&\quad P_{even}(0) \wedge \neg P_{odd}(0) \wedge P_{odd}(|X|) \\
&\wedge (\neg P_{even}(i+1) \vee \neg P_{odd}(i+1)) \\
&\wedge (P_{even}(i) \wedge X(i) \rightarrow P_{odd}(i+1)) \\
&\wedge (P_{odd}(i) \wedge X(i) \rightarrow P_{even}(i+1)) \\
&\wedge (P_{even}(i) \wedge \neg X(i) \rightarrow P_{even}(i+1)) \\
&\wedge (P_{odd}(i) \wedge \neg X(i) \rightarrow P_{odd}(i+1))
\end{aligned}$$

Proof of theorem (outline). For the if direction, let $\Psi(\bar{z}, \bar{Y})$ be a Σ_1^B-Horn formula which represents $R(\bar{z}, \bar{Y})$. Then Ψ has the form

$$\exists P_1 ... \exists P_r \forall x_1 \leq t_1 ... \forall x_s \leq t_s \phi(\bar{x}, \bar{P}, \bar{z}, \bar{Y}) \quad (4)$$

where ϕ is Horn with respect to $P_1, ..., P_r$. We outline a polynomial-time algorithm which, given numbers $a_1, ..., a_k$ (coded in unary) and finite sets $B_1, ..., B_m$ (coded by binary strings) determines whether $\Psi(\bar{a}, \bar{B})$ is true in the standard model. First note since $\bar{a}$ and $\bar{B}$ are given, each first-order term u in $\phi(\bar{x}, \bar{P}, \bar{a}, \bar{B})$ becomes a polynomial $u(x_1, ..., x_k)$. Each P_i can occur only in the context $P_i(u(\bar{x}))$ for some such term u, and the terms $t_1, ..., t_s$ bounding the x_i's evaluate to constants. The algorithm proceeds by computing for each possible $\bar{x}$-value $\bar{b} = (b_1, ..., b_s), 0 \leq b_i \leq t_i$, a simplified form $\phi[\bar{b}]$ of the instance $\phi(\bar{b}, \bar{P}, \bar{a}, \bar{B})$ of ϕ. In this form all first-order terms and all atomic formulas not involving the P_i's are evaluated, and the result is a Horn formula $\phi[\bar{b}]$ all of whose atoms are in the list $P_i(0), ..., P_i(T), i = 1, ..., r$, where T is the largest possible argument of any P_i in any instance. By taking the conjunction over all $\bar{b}$ of these instances, we obtain a propositional Horn formula PROP$[\phi, \bar{a}, \bar{B}]$. This formula is tested for satisfiability using a standard algorithm.

The proof of the only-if direction resembles the proof of Cook's theorem that SAT is NP-complete, and of Fagin's theorem of finite model theory that second-order existential formulas capture NP. The idea is to represent the computation of a Turing machine M by a two dimensional array P, where the i-th row represents the tape configuration of M (including state and scanned-symbol information) at time i. The two existential second-order quantifiers are $\exists P \exists \tilde{P}$, where $\tilde{P}$ is intended to be $\neg P$. The two universally quantified variables x_1, x_2 represent the co-ordinates of P. A crucial observation is that if M is deterministic, then the conditions on P and $\tilde{P}$ can be expressed with Horn clauses. □

Note that above proof also shows that every NP- relation can be represented by a Σ_1^B formula of the form (4), except that ϕ is not Horn.

Example. (**3COLOR**(n, E)) This is a Σ_1^B formula asserting that the graph with edge relation E on nodes $\{0, 1, ..., n-1\}$ is three-colorable. We write $E(x, y)$ like a binary relation, although it can be coded as a unary relation using the pairing function as explained above. The three colors are P, Q, and R.

$$\begin{aligned}
&\exists P \exists Q \exists R \forall x < n \forall y < n (P(x) \vee Q(x) \vee R(x)) \\
&\quad \wedge (\neg E(x, y) \vee \neg P(x) \vee \neg P(y)) \\
&\quad \wedge (\neg E(x, y) \vee \neg Q(x) \vee \neg Q(y)) \\
&\quad \wedge (\neg E(x, y) \vee \neg R(x) \vee \neg R(y))
\end{aligned}$$

This formula is Σ_1^B-Horn except for the first clause. Since graph 3-colorability is NP-complete, it cannot be represented by a Σ_1^B-Horn formula unless P= NP. This example illustrates why we cannot allow bounded first-order existential quantifiers after the universal quantifiers in Σ_1^B-Horn formulas, since the first clause could be replaced by $\exists i < 3P(i, x)$ where now $P(0, x), P(1, x), P(2, x)$ represent the three colors.

3 $\forall_1$-Horn and other second-order theories

Our second-order theories use the language $\mathcal{L}_A^2$ described in the previous section. They all share the set 2-

Robinson's theory Q axioms	
B1: $x+1 \neq 0$	B2: $x+1=y+1 \rightarrow x=y$
B3: $x+0=x$	B4: $x+(y+1)=(x+y)+1$
B5: $x \cdot 0=0$	B6:$x \cdot (y+1)=(x \cdot y)+x$
Axioms for $\leq$	
B7: $0 \leq x$	B9: $x \leq y \wedge y \leq z \rightarrow x \leq z$
B8 : $x \leq x+y$	B10 : $(x \leq y \wedge y \leq x) \rightarrow x=y$
B11: $x \leq y \vee y \leq x$	B12 : $x \leq y \leftrightarrow x < y+1$
Predecessor axiom	
B13 : $x \neq 0 \rightarrow \exists y(y+1=x)$	
Length axioms	
L1 : $X(y) \rightarrow y < \|X\|$	
L2 : $y+1=\|X\| \rightarrow X(y)$	

Table 1. The 2-$BASIC$ Axioms

$BASIC$ of axioms in Table 1, which are similar to the axioms for Zambella's theory Θ [31] and form the second-order analog of Buss's first-order axioms $BASIC$ [3]. The set 2-$BASIC$ consists essentially of the axioms for Robinson's system Q, together with axioms for $\leq$, and two axioms defining the length terms $|X|$.

The underlying logic for our theories is that of two-sorted first-order predicate calculus. Any standard proof system for predicate calculus, such as Gentzen's system LK, can be adapted to a two-sorted system simply by reinterpreting the notion of *formula* to be that defined in Section 2.

In addition to 2-$BASIC$, each system needs a comprehension scheme for some set $FORM$ of formulas.

$$FORM - \text{COMP} : \exists X \leq y \forall z < y(X(z) \leftrightarrow \Phi(z)) \quad (5)$$

Here, Φ is any formula in the set $FORM$ with no free occurrence of X.

We denote by V^i the theory axiomatized by 2-$BASIC$ and Σ^B_i-COMP. For $i \geq 0$ V^i is essentially the same as Zambella's $\Sigma^p_i - comp$ [31]. For $i \geq 1$ V^i is essentially the same as V^i_1 [20]. (The latter restricts comprehension to $\Sigma^{1,b}_0$ formulas, but allows induction on $\Sigma^{1,b}_i$ formulas. However Theorem 1 of Buss [4] shows that V^i_1 proves the $\Sigma^{1,b}_i$ comprehension axioms.) Thus for $i \geq 1$ V^i is a second-order version of S^i_2. In particular, the Σ^B_1-definable functions in V^1 are precisely the polynomial-time functions [9]. The Σ^B_1-definable functions in V^0 are the uniform $\mathtt{AC}^0$ functions [9] (called *rudimentary functions* in [31]). The first-order analog of V^0 is S^0_2 with a comprehension scheme for sharply-bounded formulas.

Definition 3.1. V_1-Horn is the theory axiomatized by 2-$BASIC$ and Σ^B_1-Horn-COMP.

Although 2-$BASIC$ does not include an explicit induction axiom, L2 asserts that a nonempty set has a largest element. This can be turned into a least number principle, from which induction follows.

Lemma 3.2. *The least number principle is a theorem of V_1-Horn, and of $V^i, i \geq 0$.*

$$LNP: 0 < |X| \rightarrow \exists x < |X|(X(x) \wedge \forall y < x \neg X(y))$$

Proof. By the comprehension schema there is a set Y such that $|Y| \leq |X|$ and for all $z < |X|$

$$Y(z) \leftrightarrow \forall i < |X|(X(i) \rightarrow z < i)$$

Thus the set Y consists of those elements smaller than every element in X. We claim that $|Y|$ satisfies the LNP for X; that is (i) $|Y| < |X|$, (ii) $X(|Y|)$ and (iii) $\forall y < |Y| \neg X(y)$. First suppose that Y is empty. Then $|Y| = 0$ by B13 and L2. By assumption $0 < |X|$, so (i) holds in this case. Also $X(0)$, since otherwise $Y(0)$ by B7 and the definition of Y, so (ii) holds. Since $\neg y < 0$ by B7 and B10 we conclude (iii) holds vacuously.

Now suppose $Y(y)$ for some y. Then $y < |Y|$ by L1, so $|Y| \neq 0$ so by B13 $|Y| = z+1$ for some z and hence $Y(z)$ by L2. Then $\neg Y(z+1)$ by L1. Thus $X(z+1)$ by B11, B12 and the definition of Y, so (ii) holds. Also $\neg X(z)$, so (i) holds. Finally (iii) holds by the definition of Y and B10. □

Lemma 3.3. *Induction on length of a string is a theorem of V_1-Horn, and of $V^i, \geq 0$.*

$$IND: (X(0) \wedge \forall y < z(X(y) \rightarrow X(y+1))) \rightarrow X(z)$$

The proof of induction is a formalization of the standard proof LNP→ IND. It can be generalized to allow induction with an arbitrary k as a basis, not just $k = 0$.

If follows from the above Lemma that each of the theories that we have presented proves an induction axiom for each formula in its comprehension scheme. In particular, for V_1-Horn we have

Corollary 3.4. *V_1-Horn proves the Σ^B_1-Horn Induction axioms.*

$$(\Phi(0) \wedge \forall y < z(\Phi(y) \rightarrow \Phi(y+1))) \rightarrow \Phi(z)$$

where Φ is any Σ^B_1-Horn formula.

Standard arguments show that induction on open formulas using axioms B1 to B13 is enough to prove simple algebraic properties of + and · such as commutativity, associativity, distributive laws, and cancellation laws involving +, ·, and $\leq$. Hence all of our theories prove these properties, and in the sequel we take them for granted. These simple properties suffice to prove that the tupling function defined in (2) and (3) is one-one, so these theories all prove

$$\langle x_1, ..., x_k \rangle = \langle x'_1, ..., x'_k \rangle \rightarrow (x_1 = x'_1 \wedge ... \wedge x_k = x'_k)$$

Notation. We use $P^{[b]}$ to denote the "b-th row" when P is being used as a 2-dimensional array. If $\phi(P)$ is a formula with no occurrence of $|P|$, then $\phi(P^{[b]})$ is obtained from $\phi(P)$ by replacing every atomic formula $P(t)$ by $P(b,t)$ (i.e. $P(\langle b,t\rangle)$: see (2)).

Other useful properties provable in V_1-Horn include a k-ary comprehension and replacement.

Lemma 3.5 (k-ary Comprehension). *If $\Phi(x_1,...,x_k)$ is a Σ_1^B-Horn formula with no free occurrence of Y, then V_1-Horn proves the k-ary comprehension formula*

$$\exists Y \leq \langle b_1,...,b_k\rangle \forall x_1 < b_1 ... \forall x_k < b_k \quad (6)$$
$$(Y(x_1,...,x_k) \leftrightarrow \Phi(x_1,...,x_k)),$$

Lemma 3.6 (Replacement). *If $\phi(y,\bar{P})$ is a Π_1^b Horn formula with respect to $\bar{P}$ and t is a term not involving y, then V_1-Horn proves*

$$\forall y < t \exists \bar{P} \phi(y,\bar{P}) \leftrightarrow \exists \bar{P} \forall y < t \phi(y, \bar{P}^{[y]})$$

where $\bar{P}^{[y]}$ is $P_1^{[y]},...,P_k^{[y]}$. Further the RHS is a Σ_1^B-Horn formula.

The replacement scheme is a corollary of the following lemma:

Lemma 3.7. *If V_1-Horn proves that $\exists P \forall y < b \Phi(y,P)$ is equivalent to some Σ_1^B-Horn then V_1-Horn proves*

$$\forall y < b \exists P \Phi(y,P) \leftrightarrow \exists P \forall y < b \Phi(y, P^{[y]}).$$

4 Formulas provably equivalent to Σ_1^B-Horn

Our goal now is to show that every Σ_0^B formula and every Σ_i^B-Horn formula, $i \in \mathbb{N}$, is provably equivalent in V_1-Horn to a Σ_1^B-Horn formula, and hence can be used in the comprehension and induction schemes. Later, we also show that the class of formulas provably equivalent to Σ_1^B-Horn is closed under $\neg, \wedge, \vee$ and bounded first-order quantification (see 5.3). We start with a simple observation.

Lemma 4.1. *If Φ_1 and Φ_2 are Σ_1^B-Horn formulas, then $\Phi_1 \wedge \Phi_2$ is logically equivalent to a Σ_1^B-Horn formula.*

Proof. Take a suitable prenex form of $\Phi_1 \wedge \Phi_2$. □

4.1 Simulating first-order bounded existential quantification

A major inconvenience of Σ_1^B-Horn formulas is lack of first-order existential quantifiers. In general we cannot allow such quantifiers without increasing the apparent expressive power of the formulas, as pointed out in the 3-colorability example. However, it is possible to introduce bounded existential quantifiers in some contexts.

Notation. If P is a second-order variable, then $\tilde{P}$ denotes a second-order variable whose intended interpretation is $\neg P$.

We now introduce the Horn formulas SEARCH_k, which are Π_1^b Horn with respect to all of their second-order variables and which will allow a Σ_1^B-Horn formula to represent $\exists z < b X(\bar{y},z)$. Assuming that $\tilde{X} \leftrightarrow \neg X$, $\text{SEARCH}_k(\bar{b},b,S,\tilde{S},X,\tilde{X})$ asserts that $S(\bar{y},i)$ holds iff $X(\bar{y},z)$ holds for some $z < i$, where $\bar{b}$ stands for $b_1,...,b_k$, and $\bar{y}$ stands for $y_1,...,y_k$. We use $\bar{y} < \bar{b}$ for $y_1 < b_1 \wedge ... \wedge y_k < b_k$.

Definition 4.2. For each $k \geq 1$ $\text{SEARCH}_k(\bar{b},b,S,\tilde{S},X,\tilde{X})$ is the Π_1^b Horn formula

$$\begin{aligned}\forall \bar{y} < \bar{b} \forall i < b (&\neg S(\bar{y},0) \wedge \tilde{S}(\bar{y},0)) \\ &\wedge (\neg S(\bar{y},i+1) \vee \neg \tilde{S}(\bar{y},i+1)) \\ &\wedge (S(\bar{y},i) \rightarrow S(\bar{y},i+1)) \\ &\wedge (X(\bar{y},i) \rightarrow S(\bar{y},i+1)) \\ &\wedge (\tilde{S}(\bar{y},i) \wedge \tilde{X}(\bar{y},i) \rightarrow \tilde{S}(\bar{y},i+1))\end{aligned}$$

We can prove in V_1-Horn that this definition of SEARCH corresponds to a bounded existential quantifier in the above limited sense.

4.2 The Σ_0^B formulas are provably equivalent to Σ_1^B-Horn

Consider a Σ_0^B formula $Q_1 y_1 < b_1 ... Q_k y_k < b_k \phi(\bar{y})$, where each Q_i is either $\forall$ or $\exists$. We can shows how to conjoin copies of SEARCH(...) to define arrays $S_0,...,S_k$ such that $S_i(y_1,...,y_{k-i}) \leftrightarrow Q_{k-i+1} y_{k-i+1} < b_{k-i+1} \phi(\bar{y})$. These are used to form an equivalent Σ_1^B-Horn formula.

The proof of this fact proceeds by induction. For the base case, we define $S_0(0) \equiv \phi(\bar{y})$ and $\tilde{S}_0(0) \equiv \neg\phi(\bar{y})$ (we can negate a quantifier-free formula). For the induction step, to get the arrays S_{i+1} and S_{i+1} we search the values of S_i and $\tilde{S}_i$ for either a witness or counterexample, based on whether the i^{th} quantifier of the original formula is $\exists$ or $\forall$.

Corollary 4.3. *Every Σ_0^B formula is provably equivalent in V_1-Horn to a Σ_1^B-Horn formula.*

4.3 Collapse of V-Horn to V_1-Horn

Grädel [14] showed that it is possible to represent a $SO\exists$-Horn formula preceded by alternating SO quantifiers by a $SO\exists$-Horn formula, which implies the collapse of SO-Horn hierarchy to $SO\exists$-Horn. We can formalize Grädel's proof in V_1-Horn, showing that Σ_1^B-Horn formulas are closed under second-order quantification. That is, a Σ_1^B-Horn formula preceded by a sequence of (possibly alternating) second-order quantifiers is equivalent (provably in V_1-Horn) to a Σ_1^B-Horn formula.

5 Encoding the Horn SAT algorithm by a Σ_1^B-Horn formula

Here we show that a run of the Horn satisfiability algorithm described in the proof of Theorem 2.3 can be represented by a Σ_1^B-Horn formula RUN. This result is needed for sections 6 and 7. A simple corollary is that the negation of a Σ_1^B-Horn formula is provably equivalent to a Σ_1^B-Horn formula. In other words, V_1-Horn proves that P is closed under complementation.

Theorem 5.1. *Let Φ be a Σ_1^B-Horn formula which does not involve R or $\tilde{R}$. Then there is a formula $\text{RUN}_\Phi(R,\tilde{R})$ whose free variables include those of Φ in which the only atomic subformulas involving R and $\tilde{R}$ are $R(0)$ and $\tilde{R}(0)$ and such that $\exists R\exists\tilde{R}\text{RUN}_\Phi(R,\tilde{R})$ is a Σ_1^B-Horn formula and V_1-Horn proves the following:*

(i) $\exists R\exists\tilde{R}\ \text{RUN}_\Phi(R,\tilde{R})$

(ii) $\text{RUN}_\Phi(R,\tilde{R}) \rightarrow [(R(0)\leftrightarrow\Phi)\wedge(\tilde{R}(0)\leftrightarrow\neg\Phi)]$

In the proof of this theorem, it is sufficient to consider only Σ_1^B-Horn formulas with one existential second-order quantifier.

Corollary 5.2. *If Φ is Σ_1^B-Horn, then $\neg\Phi$ is provably equivalent in V_1-Horn to a Σ_1^B-Horn formula NEG_Φ.*

Corollary 5.3. *The class of formulas provably equivalent in V_1-Horn to a Σ_1^B-Horn formula is closed under $\neg$, $\wedge$, $\vee$, and bounded first-order quantification.*

This follows from lemmas 3.6 and 4.1, and corollary 5.2.

Theorem 5.1 can be generalized to the case in which arrays $R(\bar{y})$ and $\tilde{R}(\bar{y})$ code values of $\Phi(\bar{y})$ and $\neg\Phi(\bar{y})$.

Corollary 5.4. *Let $\Phi(\bar{y})$ be a Σ_1^B-Horn formula which does not involve R or $\tilde{R}$. Then there is a formula $\text{RUN}_{\Phi(\bar{y})}(\bar{b},R,\tilde{R})$ which does not have $\bar{y}$ free but whose free variables include any other free variables of Φ such that $\exists R\exists\tilde{R}\text{RUN}_{\Phi(\bar{y})}(R,\tilde{R})$ is a Σ_1^B-Horn formula and V_1-Horn proves the following:*

(i) $\exists R\exists\tilde{R}\ \text{RUN}_{\Phi(\bar{y})}(\bar{b},R,\tilde{R})$

(ii) $\text{RUN}_{\Phi(\bar{y})}(\bar{b},R,\tilde{R}) \rightarrow$
$\forall\bar{y}<\bar{b}[(R(\bar{y})\leftrightarrow\Phi(\bar{y}))\wedge(\tilde{R}(\bar{y})\leftrightarrow\neg\Phi(\bar{y}))]$

The algorithm we wish to represent has two main steps (see the proof of Theorem 2.3): First create a propositional Horn formula $\text{HORN}[\Phi]$ (which depends on the values for the free variables in Φ), and second apply the Horn Sat algorithm to determine whether $\text{HORN}[\Phi]$ is satisfiable. We encode Φ in $\text{HORN}[\Phi]$ using an array Q, and we will present a Σ_1^B-Horn formula $\text{PROP}_\Phi(Q,\tilde{Q})$ which defines this array and its negation. Besides the indicated free variables, PROP_Φ also has as free variables the free variables of Φ.

For the second step we present a Σ_1^B-Horn formula $\text{HORNSAT}(a,b,Q,\tilde{Q},R,\tilde{R})$ (with all free variables indicated) which is independent of Φ and which sets the result variable $R(0)$ true iff $\text{HORN}[\Phi]$ is satisfiable. The encoding Q consists of three parts: $C(x,v)$, $D(x,v)$ and $V(x)$. The first two assert that a clause x contains a positive (resp. negative) literal v; the last states that the clause x is true. All formulas defining Q and $\tilde{Q}$ are Σ_0^B.

We can now choose $\text{RUN}_\Phi(R,\tilde{R})$ to be a Σ_1^B-Horn formula such that

$$\text{RUN}_\Phi(R,\tilde{R}) \leftrightarrow \exists\bar{Q}[\text{PROP}_\Phi(\bar{Q})\wedge\text{HORNSAT}(\hat{a},\hat{b},\bar{Q},R,\tilde{R})]$$

In fact we take $\text{RUN}_\Phi(R,\tilde{R})$ to be a suitable prenex form of the right hand side.

5.1 Definition of $\text{PROP}_\Phi(Q,\tilde{Q})$

We define three Σ_0^B formulas $\psi_C(x,v),\psi_D(x,v),\psi_V(x)$ which characterize the three arrays C,D,V.

Lemma 5.5. $\text{PROP}_\Phi(\bar{Q})$ *can be defined in such a way that $\exists\bar{Q}\text{PROP}_\Phi(\bar{Q})$ is Σ_1^B-Horn and V_1-Horn proves*

(i) $\exists\bar{Q}\text{PROP}_\Phi(\bar{Q})$

(ii) $\text{PROP}_\Phi(\bar{Q}) \rightarrow \forall v<\hat{a}\forall x<\hat{b}$
$[(C(x,v)\leftrightarrow\psi_C(x,v))\wedge(\tilde{C}(x,v)\leftrightarrow\neg\psi_C(x,v))$
$\wedge(D(x,v)\leftrightarrow\psi_D(x,v))\wedge(\tilde{D}(x,v)\leftrightarrow\neg\psi_D(x,v))$
$\wedge(V(x)\leftrightarrow\psi_V(x))\wedge(\tilde{V}(x)\leftrightarrow\neg\psi_V(x))]$

5.2 Definition of $\text{HORNSAT}(a,b,Q,\tilde{Q},R,\tilde{R})$

Although the Horn satisfiability algorithm is easy to describe informally, it is not straightforward to formalize in V_1-Horn. The propositional Horn satisfiability problem is complete for P, [16], and hence cannot be represented by a Σ_0^B formula.

The algorithm represented by $\text{HORNSAT}(a,b,Q,\tilde{Q},R,\tilde{R})$ attempts to find a satisfying assignment to the Horn formula $\text{HORN}[\Phi]$ described by the parameters a,b,Q. This is done by filling in an array $T(t,v)$, where $T(t,v)$ is the truth value assigned to the atom $P(v)$ after step t, $0\le t,v<a$. Initially $T(0,v)$ is false, and at step $t+1$, $T(t+1,v)$ sets each $P(v)$ true such that $P(v)$ occurs positively in some clause not satisfied after step t. Once $P(v)$ is set true, it is never changed to false.

Defining TDEF to be the formula encoding the truth assignment array T and SAT a formula stating that the resulting truth assignment $T^{[a]}$ satisfies $\text{HORN}[\Phi]$, the rest of the

algorithm is encoded by

$$\begin{aligned}\text{HORNSAT}(a,b,\bar{Q},R,\tilde{R}) \equiv \\ \exists T \exists \tilde{T}[\text{TDEF}(a,b,\bar{Q},T,\tilde{T}) \wedge \\ \exists \bar{W}\,\text{SAT}^*(1,R,\tilde{R},\bar{W},\bar{Q},T^{[a]},\tilde{T}^{[a]})]\end{aligned} \quad (7)$$

Lemma 5.6 (Correctness of HORNSAT). *V_1-Horn proves*

$$\begin{aligned}\text{HORNSAT}(a,b,\bar{Q},R,\tilde{R}) \wedge \text{NEG} \rightarrow \\ (R(0) \leftrightarrow \exists T_1 \text{SAT}(a,b,\bar{Q},T_1)) \\ \wedge(\tilde{R}(0) \leftrightarrow \neg\exists T_1 \text{SAT}(a,b,\bar{Q},T_1)),\end{aligned}$$

where NEG *states that* $Q \leftrightarrow \neg\tilde{Q}$.

Full details can be found in [7].

6 Equivalence of V_1-Horn, P-def and QPV

The first-order theory QPV (called PV1 in [20]) has function symbols for all polynomial-time computable functions, and the axioms include defining equations for these functions (based on Cobham's Theorem) and induction on the length of numbers. The theory has been extensively studied [8, 3, 12, 20, 10] and shown to robustly capture the notion of "polynomial-time reasoning". Zambella's [31] theory P-def is a second-order version of QPV, and can shown to be equivalent to QPV by the method of RSUV isomorphism (see [20]). We show that V_1-Horn is equivalent in power to P-def. This implies that V_1-Horn is equivalent in power to QPV, but is most likely not as powerful as S_2^1 (see Section 1).

We add function symbols to V_1-Horn by defining their bit graphs by Σ_1^B-Horn formulas, obtaining a system V_1-Horn(FP) of the same power as V_1-Horn. Then, we prove the equivalence (provable both in P-def and V_1-Horn(FP)) of functions defined with Σ_1^B-Horn formulas and function defined by Cobham's theorem. Finally, we show that the classes of theorems of V_1-Horn(FP) and theorems of P-def coincide. The main result of this section is:

Theorem 6.1. *P-def is a conservative extension of V_1-Horn.*

7 Finite Axiomatizability

Here we show that both V^0 and V_1-Horn are finitely axiomatizable, and that the $\forall\Sigma_1^B$ consequences of V_1-Horn and the $\forall\Sigma_1^b$ consequences of S_2^1 are each finitely axiomatizable. (Theorem 10.1.2 of [20] states that the $\forall\Sigma_j^b$ consequences of S_2^i are finitely axiomatizable for $j \geq 2$ and $i \geq 1$.)

Since V^0 defines the uniform AC^0 functions, it seems plausible that V_1-Horn could be axiomatized by V^0 together with a formula expressing the comprehension axiom for some predicate which is complete for P under uniform AC^0 reductions. Hence the finite axiomatizability of V_1-Horn should follow from that for V^0. In our proof of Theorem 7.5 below, that predicate is the Horn satisfiability problem, which is complete for P [16].

Theorem 7.1. *V^0 is finitely axiomatizable.*

Proof. We must show that all Σ_0^B-COMP axioms follow from finitely many theorems of V^0 (see section 3).

Let $2 - BASIC^+$ (or simply B^+) denote the $2 - BASIC$ axioms along with finitely many theorems of V^0 asserting basic properties of $+$ and $\cdot$ such as commutativity, associativity, distributive laws, and cancellation laws involving $+$, $\cdot$, and $\leq$. These can be proved from the $2 - BASIC$ axioms by induction on Σ_0^B formulas, as discussed in Section 3.

It suffices to show that k-ary comprehension (6) for all Σ_0^B formulas follow from B^+ and finitely many such comprehension instances. We use the notation $\Phi[\bar{a},\bar{Q}](\bar{x})$ to indicate that the Σ_0^B formula Φ can contain the free variables $\bar{a},\bar{Q}$ in addition to $\bar{x} = x_1,...,x_k$. Then $COMP_\Phi(\bar{a},\bar{Q},\bar{b})$ denotes the comprehension formula

$$\exists Y \leq \langle b_1,...,b_k\rangle \forall x_1 < b_1 ... \forall x_k < b_k (Y(\bar{x}) \leftrightarrow \Phi(\bar{x}))$$

We can show that there are only 12 formulas $\Phi_1,\ldots,\Phi_{12}$ for which we need instances $COMP_\Phi$ of comprehension scheme. For example, Φ_1, Φ_2 and Φ_3 are:

$$\begin{aligned}\Phi_1(x_1,x_2) &\equiv \exists y \leq x_1(x_1 = \langle x_2, y\rangle) \\ \Phi_2(x_1,x_2) &\equiv \exists z \leq x_1(x_1 = \langle z, x_2\rangle) \\ \Phi_3[Q_1,Q_2](x_1,x_2) &\equiv \exists y \leq x_1(Q_1(x_1,y) \wedge Q_2(y,x_2))\end{aligned}$$

In the following lemmas, we abbreviate $COMP_{\Phi_i}(...)$ by C_i. The lemmas state that projection, terms and finally atomic formulas can be defined using finitely many axioms of V^0.

Lemma 7.2. *For each $k \geq 2$ and $1 \leq i \leq k$ let*

$$\begin{aligned}\Psi_{ik}(y,z) \equiv \exists x_1 \leq y \ldots \exists x_{i-1} \leq y \exists x_{i+1} \leq y \ldots \exists x_k \leq y \\ (y = \langle x_1,\ldots,x_{i-1},z,x_{i+1},\ldots,x_k\rangle)\end{aligned}$$

Then

$$B^+, C_1, C_2, C_3 \vdash COMP_{\Psi_{ik}}$$

Lemma 7.3. *Let $t(\bar{x})$ be a term which in addition to variables $\bar{x}$ may involve other variables $\bar{a},\bar{Q}$. Let $\Psi_t[\bar{a},\bar{Q}](\bar{x},y) \equiv y = t(\bar{x})$. Then*

$$B^+, C_1, ..., C_6 \vdash COMP_{\Psi_t}(\bar{a},\bar{Q},\bar{b},d)$$

Lemma 7.4. *Let $t_1(\bar{x}), t_2(\bar{x})$ be terms with variables among $\bar{x},\bar{a},\bar{Q}$. Suppose*

$$\Psi_1[\bar{a},\bar{Q}](\bar{x}) \equiv t_1(\bar{x}) = t_2(\bar{x})$$
$$\Psi_2[\bar{a},\bar{Q}](\bar{x}) \equiv t_1(\bar{x}) \le t_2(\bar{x})$$
$$\Psi_3[\bar{a},\bar{Q},X](\bar{x}) \equiv X(t_1(\bar{x}))$$

Then $B^+, C_1, ..., C_9 \vdash COMP_{\Psi_i}$, *for* $i = 1, 2, 3$.

Now we can complete the proof of the theorem. Lemma 7.4 takes care of the case when Φ is an atomic formula. Let

$$\Phi_{10}[Q](x) \equiv \neg Q(x)$$
$$\Phi_{11}[Q_1,Q_2](x) \equiv Q_1(x) \wedge Q_2(x)$$
$$\Phi_{12}[Q,c](x) \equiv \forall y \le c Q(x,y)$$

Now by repeated applications of $COMP_{\Phi_{10}}$ and $COMP_{\Phi_{11}}$ we handle the case in which Φ is quantifier-free.

Now suppose $\Phi(\bar{x}) \equiv \forall y \le t(\bar{x})\phi(\bar{x},y)$. We assume as an induction hypothesis that we can define Q satisfying

$$\forall \bar{x} < \bar{b} \forall y < t(\bar{b}) + 1 [Q(\bar{x},y) \leftrightarrow (y \le t(\bar{x}) \rightarrow \phi(\bar{x},y)]$$

Then $COMP_\Phi(\bar{b})$ follows from $COMP_{\Phi_{12}}(Q,c,b)$ with $c \leftarrow t(\bar{b})$ and $b \leftarrow \langle b_1, ..., b_k \rangle$. □

Theorem 7.5. *V_1-Horn is finitely axiomatizable.*

Proof. It suffices to show that Corollary 5.4 (i) and (ii) can be proved for any Σ^B_1-Horn formula $\Phi(y)$ using finitely many theorems of V_1-Horn as axioms. We first will show how to do this for Theorem 5.1 (i) and (ii), and then explain how to modify the proof to get the corollary.

First note that for each Σ^B_1-Horn formula Φ we can define a version of PROP_Φ such that (i) and (ii) in Lemma 5.5 are theorems of V^0. Thus we include the finite set of axioms for V^0 from Theorem 7.1 among the finite axioms for V_1-Horn. The proof of Theorem 5.1 depends on Lemma 5.5 (which we have established) and some properties of HORNSAT. Since HORNSAT is independent of Φ, we can take these properties as axioms.

To generalize the proof of Theorem 5.1 in order to prove Corollary 5.4, we incorporate the variable y in $\Phi(y)$ as an argument of each of the arrays $C, D, V, \tilde{C}, \tilde{D}, \tilde{V}$ to define the formula $\text{PROP}_\Phi(y)$ in a modified Lemma 5.5. Then y is not free in $\text{PROP}_\Phi(y)$ (although it could be free in PROP_Φ). The definition (7) of HORNSAT is modified so that the parameter y is incorporated as an argument of each of the arrays $R, \tilde{R}, T, \tilde{T}$. Then Corollary 5.4 follows in the same way as Theorem 5.1. □

Theorem 7.6. *V_1-Horn is axiomatized by its $\forall\Sigma^B_1$ consequences.*

Proof. It suffices to show that each Σ^B_1-Horn comprehension axiom is a consequence of $\forall\Sigma^B_1$ theorems of V_1-Horn. First we show that the second-order quantifiers in Σ^B_1-Horn formulas (1) can be bounded. That is, for each Σ^B_1-Horn formula Φ there is a Σ^B_1 formula Φ^B such that $\forall\Sigma^B_1 V_1\text{-Horn} \vdash (\Phi \leftrightarrow \Phi^B)$. To construct Φ^B replace each second-order quantifier $\exists P$ in Φ by a bounded quantifier $\exists P \le t$, where t is a provable upper bound on all terms u such that $P(u)$ occurs in Φ. The equivalence of Φ and Φ^B requires only Ψ-COMP instances for formulas Ψ with no second-order quantifiers, and these instances are $\forall\Sigma^B_1$ formulas.

The comprehension axiom (5) for $\Phi(z)$ follows from Corollary 5.4 (i) and (ii). The Σ^B_1 form of (i) we need is

$$\exists R \le y \exists \tilde{R} \le y \, \text{RUN}'_{\Phi(z)}(y, R, \tilde{R})$$

where $\text{RUN}'_{\Phi(z)}$ has suitable bounds on its second-order quantifiers. For (ii) we do not need the clause involving $\tilde{R}$. If we replace Φ by Φ^B then a suitable prenex form of the result is $\forall\Sigma^B_1$. □

Corollary 7.7. *The $\forall\Sigma^B_1$ consequences of V_1-Horn are finitely axiomatizable. The $\forall\Sigma^b_1$ consequences of S^1_2 are finitely axiomatizable.*

Proof. The first sentence follows by compactness from Theorems 7.6 and 7.5. Since V^1 is $\forall\Sigma^B_1$ conservative over P-def [31], it follows from Theorem 6.1 that the $\forall\Sigma^B_1$ consequences of V^1 and of V_1-Horn are the same, and hence are finitely axiomatizable. The second sentence of the Corollary is equivalent to asserting that the $\forall\Sigma^B_1$ consequences of V^1 are finitely axiomatizable, by the RSUV isomorphism. □

8. Conclusion

The original motivation for this paper was to make a connection between descriptive complexity and bounded arithmetic. Specifically we use Grädel's theorem that a predicate is polynomial-time iff it corresponds to the finite models of some second-order Horn formula, and define a second-order theory based on a comprehension axiom scheme essentially over the second-order Horn formulas. The resulting theory V_1-Horn turns out to have the same power as the previously-defined theories QPV and P-def but the proof of equivalence is nontrivial and requires formalizing the Horn satisfiability algorithm in V_1-Horn. Unlike QPV and P-def, our theory V_1-Horn turns out to be finitely axiomatizable, and this has consequences for the important theory S^1_2.

It seems plausible that characterizations of other complexity classes in descriptive complexity can be used to define related theories. In particular, Grädel[15] uses second-order Krom formulas to characterize NL (nondeterministic log space), and this might serve a basis for a theory of log space reasoning.

Although we do not exploit them in this paper, bounded arithmetic has important connections with propositional

proof complexity (see [20]). The main goal of the latter is to establish super-polynomial lower bounds on the lengths of proofs in various propositional proof systems. (If this could be done for all "reasonable" such systems then NP $\neq$ *co*NP and hence NP $\neq$ P [11].) [8] showed that every theorem of PV can be expressed as a family of tautologies with polynomial size proofs in a so-called Extended-Frege proof system. A host of similar results has been proved since. In the case of some weak theories T the corresponding propositional proof system is sometimes weak enough that super-polynomial lower bounds are provable, and then independence results for T follow [24]. We know indirectly from [8] that the Σ_0^B theorems of V_1-Horn translate into tautology families with polynomial-size Extended-Frege proofs. It might be instructive to carry out this translation directly, possibly shedding light on the central and very difficult problem of proving superpolynomial lower bounds for Extended-Frege systems.

References

[1] D. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274 – 30, 1990.

[2] S. Buss. Collection of papers. URL: "ftp://euclid.ucsd.edu/pub/sbuss/research/".

[3] S. Buss. *Bounded Arithmetic*. Bibliopolis, Naples, 1986.

[4] S. Buss. Axiomatizations and conservation results for fragments of bounded arithmetic. *Contemporary Mathematics*, 106:57–84, 1990.

[5] S. Buss. Relating the bounded arithmetic and polynomial time hierarchies. *Annals of Pure and Applied Logic*, 75:67–77, 1995.

[6] S. Buss, editor. *Handbook of Proof Theory*. Elsevier, Amsterdam, 1998.

[7] S. Cook and A. Kolokolova. A second-order system for polynomial-time reasoning based on Graedel's theorem. *Electronic Colloquium on Computational Complexity (ECCC)*, TR01-024, 2001.

[8] S. A. Cook. Feasibly constructive proofs and the propositional calculus. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, pages 83 –97, 1975.

[9] S. A. Cook. CSC 2429S: Proof Complexity and Bounded Arithmetic. Course notes, URL: "http://www.cs.toronto.edu/~sacook/csc2429h", Spring 1998.

[10] S. A. Cook. Relating the provable collapse of P to NC^1 and the power of logical theories. *DIMACS series in Discrete mathematics and theoretical computer science*, 39:73–91, 1998.

[11] S. A. Cook and A. R. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36 –50, 1979.

[12] S. A. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103 – 200, 1993.

[13] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation, SIAM-AMC proceedings*, 7:43–73, 1974.

[14] E. Grädel. The Expressive Power of Second Order Horn Logic. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science STACS '91, Hamburg 1991*, volume 480 of *LNCS*, pages 466–477. Springer-Verlag, 1991.

[15] E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.

[16] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation*. Oxford University Press, 1995.

[17] P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic*. Springer, Berlin, 1998.

[18] N. Immerman. Relational queries computable in polytime. *Information and Control*, 68:86 –104, 1986.

[19] N. Immerman. *Descriptive complexity*. Springer Verlag, New York, 1999.

[20] J. Krajiček. *Bounded Arithmetic, Propositional Logic, and Complexity Theory*. Cambridge University Press, New York, USA, 1995.

[21] J. Krajiček, P. Pudlák, and G. Takeuti. Bounded arithmetic and the polynomial time hierarchy. *Annals of Pure and Applied Logic*, 52:143–153, 1991.

[22] D. Leivant. Characterization of complexity classes in higher-order logic. In *Proceedings of the Second Annual Conference on Structure in Complexity Theory*, pages 203–217, 1987.

[23] D. Leivant. Descriptive characterizations of computational complexity. *Journal of Computer and System Sciences*, 39:51–83, 1989.

[24] J. Paris and A. Wilkie. Counting problems in bounded arithmetics. In *Methods in mathematical logic*, volume LNM 1130, pages 317 – 340. Springer Verlag, 1985.

[25] A. Razborov. An equivalence between second-order bounded domain bounded arithmetic and first-order bounded arithmetic. In P. Clote and J. Krajiček, editors, *Arithmetic, proof theory and computational complexity*, pages 247–277. Clarendon Press, Oxford, 1993.

[26] A. Razborov. Bounded arithmetic and lower bounds in boolean complexity. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 344–386. Birkhäuser, 1995.

[27] U. Schöning and R. Pruim. *Gems of theoretical computer science*. Springer, Berlin, 1998.

[28] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[29] G. Takeuti. RSUV isomorphism. In P. Clote and J. Krajiček, editors, *Arithmetic, proof theory and computational complexity*, pages 364–386. Clarendon Press, Oxford, 1993.

[30] M. Vardi. Complexity of relational query languages. *Information and Control*, 68:137 –146, 1986.

[31] D. Zambella. Notes on polynomially bounded arithmetic. *The Journal of Symbolic Logic*, 61(3):942–966, 1996.

The Crane Beach Conjecture

DAVID A. MIX BARRINGTON *
Computer Science Department
University of Massachusetts
barring@cs.umass.edu

NEIL IMMERMAN †
Computer Science Department
University of Massachusetts
immerman@cs.umass.edu

CLEMENS LAUTEMANN
Institut für Informatik
Johannes Gutenberg-Universität Mainz
cl@informatik.uni-mainz.de

NICOLE SCHWEIKARDT
Institut für Informatik
Johannes Gutenberg-Universität Mainz
nisch@informatik.uni-mainz.de

DENIS THÉRIEN ‡
School of Computer Science
McGill University
denis@cs.mcgill.ca

Abstract

A language L over an alphabet A is said to have a neutral letter *if there is a letter $e \in A$ such that inserting or deleting e's from any word in A^* does not change its membership (or non–membership) in L.*

The presence of a neutral letter affects the definability of a language in first–order logic. It was conjectured that it renders all numerical predicates apart from the order predicate useless, i.e., that if a language L with a neutral letter is not definable in first–order logic with linear order, then it is not definable in first–order logic with any set $\mathcal{N}$ of numerical predicates.

We investigate this conjecture in detail, showing that it fails already for $\mathcal{N} = \{+, *\}$, or, possibly stronger, for any set $\mathcal{N}$ that allows counting up to the m times iterated logarithm, $\lg^{(m)}$, for any constant m.

On the positive side, we prove the conjecture for the case of all monadic numerical predicates, for $\mathcal{N} = \{+\}$, for the fragment $BC(\Sigma_1)$ of first–order logic, and for binary alphabets.

*Supported by NSF grant CCR-9988260.
†Supported by NSF grant CCR-9877078.
‡Supported by NSERC and FCAR.

1 Introduction

Logicians have long been interested in the relative expressive power of different logical formalisms. In the last twenty years, these investigations have also been motivated by a close connection to computational complexity theory — most computational complexity classes have been given characterisations as finite model classes of appropriate logics, cf. [Imm98]. In these investigations it became apparent that in order to describe computation over a finite structure, a formula has to be able to refer to some linear order of the elements of this structure. Given such an order, the universe of the structure, i.e., the set of its elements, can be identified with an initial segment of the natural numbers. In a logic with the capability to express induction we can then define predicates for arithmetical operations such as addition or multiplication on the universe, and use them in order to describe operations on time or memory locations. In weak logics, however, e.g., first–order logic, defining an order relation does not automatically make arithmetic available. In fact, even over strings, the expressive power of first–order logic varies considerably, depending on the set of numerical predicates that can be used.

As an example, if the order is the only numerical relation then the only *regular* languages that can be defined in first–order logic are the star–free languages. If, however, for every $p \in \mathbb{N}$ we have available the predicate mod_p (which holds for a number m iff $m \equiv 0 \pmod p$) then we can express regular languages that are not star–free,

0-7695-1281-X/01 $10.00 © 2001 IEEE

such as $(000 + 001)^*$. In fact, with these predicates we can express *all* the first–order definable regular languages, cf. [Str94]. Thus, even very powerful relations (arithmetical relations, or even undecidable ones) are of no further help in defining regular languages. On the other hand, with addition, we can express languages that are not regular, such as $\{0^n 1^n \,/\, n \in \mathbb{N}\}$.

First-order logic with varying numerical predicates can also be thought of as specifying circuit complexity classes with varying *uniformity conditions* [BIS90]. The language defined by a first-order formula is naturally computed by a family of boolean circuits with constant depth, polynomial size, and unbounded fan-in (called "AC^0 circuits"). The power of such a family depends in part on the sophistication of the connections among the nodes. A formula with only simple numerical predicates leads to a circuit family where these connections are easily computable. These are called "uniform circuits", and how uniform they are is quantified by the computational complexity of a language describing the connections. A formula with arbitrary numerical predicates leads to a circuit family with arbitrary connections — the set of languages so describable is called "non-uniform AC^0".

There are languages, such as the PARITY language, for which we can prove no AC^0 circuit exists [Ajt83, FSS84]. A major open problem in complexity theory is to develop methods for showing languages to be outside of uniform circuit complexity classes even if they are in the corresponding non-uniform class. This is an additional motivation for the study of the expressive power of first-order logic with various numerical predicates, as this provides a parametrization of various versions of "uniform AC^0".

In an attempt to obtain a better understanding of this expressive power, Thérien considered the concept of a *neutral letter* for a language L, i.e., a letter e that can be inserted into or deleted from a string without affecting its membership in L. Since, in the presence of such a letter, membership in L cannot depend on specific (combinations of) letters being in specific (combinations of) positions, it seemed conceivable that neutral letters would render all numerical predicates, except for the order, useless. With this in mind, Thérien proposed what was later dubbed the *Crane Beach Conjecture*:

> If a language with a neutral letter can be defined in first–order logic using some set $\mathcal{N}$ of numerical predicates then it can be so defined using only the order relation.

One particular example of a language with a neutral letter is PARITY, consisting precisely of those 0–1–strings in which 1 occurs an even number of times. PARITY is not definable in first–order logic – no matter what numerical predicates are used (cf. [Ajt83, FSS84]). The Crane Beach conjecture would imply this result, since PARITY is a regular language known not to be star–free.

In this paper, we investigate the Crane Beach conjecture in detail. We first show that in general it is not true — in fact, it already fails for $\mathcal{N} = \{+, *\}$. However, we also show that the conjecture is true in a number of interesting special cases, including the case of addition, i.e., when $\mathcal{N} = \{+\}$.

This work is closely related to a line of research in data base theory which is concerned with so–called *collapse results* (cf. [BL00]). Here one considers a finite data base embedded in some infinite, ordered domain, and then looks at *locally generic* queries, i.e., queries which are invariant under monotone injections of the data base universe into the larger domain. In this setting, a language with a neutral letter is the special case of a locally generic (Boolean) query over monadic databases with background structure $\langle \mathbb{N}, \mathcal{N} \rangle$, and the conjecture then can be translated into a collapse for first–order logic.
We will come back to this in connection with Theorem 3.12.

Acknowledgements
We are indebted to Thomas Schwentick for bringing the data base theory connection to our attention. He also took an active part in many discussions on the subject of this paper. In particular, the first proof of Theorem 3.9 was partly due to him. The first author in particular would like to thank Eric Allender, Pierre McKenzie, and Howard Straubing for valuable discussions on this topic, many of which occurred at a Dagstuhl workshop in March 1997. Much important work on this topic also occurred at various McGill Invitational Workshops on Complexity Theory, particularly on excursions to Crane Beach, St. Philip, Barbados.

2 Preliminaries

2.1 First–Order Logic

A *signature* is a set σ containing finitely many relation, or predicate, symbols, each with a fixed arity. A σ–structure $\mathfrak{A} = \langle U^{\mathfrak{A}}, \sigma^{\mathfrak{A}} \rangle$ consists of a set $U^{\mathfrak{A}}$, called the *universe* of $\mathfrak{A}$ and a set $\sigma^{\mathfrak{A}}$ that contains an interpretation $R^{\mathfrak{A}} \subseteq (U^{\mathfrak{A}})^k$ for each k–ary relation symbol $R \in \sigma$.
In this paper, we are concerned almost exclusively with first–order logic over finite strings. In this context, for an alphabet A we use the signature $\sigma_A := \{Q_a \,/\, a \in A\}$ and identify a string $w = w_1 \cdots w_n \in A^*$ with the structure $w = \langle \{1, \ldots, n\}, \sigma_A^w \rangle$, where $\sigma_A^w = \{Q_a^w \,/\, a \in A\}$ and $Q_a^w = \{i \leq n \,/\, w_i = a\}$, i.e, $i \in Q_a^w \iff w_i = a$, for all $a \in A$.
In addition to the predicates Q_a we also have *numerical predicates*. A k–ary numerical predicate P has, for every

$n \in \mathbb{N}$, a fixed interpretation $P_n \subseteq \{1, \ldots, n\}^k$. Our prime example of a numerical predicate is the linear order relation $\leq$. Where we see no danger of confusion (i.e., almost everywhere) we will not distinguish notationally between a predicate and its interpretation.

An *atomic σ–formula* is either of the form $x_1 = x_2$, or $P(x_1, \ldots, x_k)$, where $x_1, x_2, \ldots, x_k$ are variables and $P \in \sigma$ is a k–ary predicate symbol. First–order σ–formulas are built from atomic σ–formulas in the usual way, using Boolean connectives $\wedge$, $\vee$, $\neg$, etc. and universal $(\forall x)$ and existential $(\exists x)$ quantifiers.

For every alphabet A, and every set $\mathcal{N}$ of numerical predicates, we will denote the set of first–order $\sigma_A \cup \mathcal{N}$–formulas by $FO[\mathcal{N}]$. We define semantics of first–order formulas in the usual way. In particular, for a string $w \in A^*$ and a formula $\varphi \in FO[\mathcal{N}]$ without free variables (i.e., variables not bound by a quantifier), we will write $w \models \varphi$ if φ holds on the string w. If $x_1, \ldots, x_k$ are the free variables of φ, and if $p_1, \ldots, p_k \leq |w|$, $w \models \varphi(p_1, \ldots, p_k)$ indicates that φ holds on the string w with x_i interpreted as p_i, for every $i \leq k$.

Every formula $\varphi \in FO[\mathcal{N}]$ without free variables defines the set L_φ of those A–strings which satisfy φ. We say that a language $L \subseteq A^*$ is *definable in $FO[\mathcal{N}]$*, and write $L \in FO[\mathcal{N}]$, if $L = L_\varphi$, for some $\varphi \in FO[\mathcal{N}]$. We will use analogous notation for subsets of $FO[\mathcal{N}]$, in particular, we will consider the set $\Sigma_1[\mathcal{N}]$ of formulas which are of the form $\exists x_1 \cdots \exists x_r \psi$, for some quantifier–free $\psi \in FO[\mathcal{N}]$, and its Boolean closure, $BC(\Sigma_1[\mathcal{N}])$. (One can define a complete hierarchy of classes $\Sigma_i[\mathcal{N}]$ and $\Pi_i[\mathcal{N}]$ along with their Boolean closures, using the hierarchy of first-order formulas given by the number of quantifier alternations. But in this paper we will have need only for $BC(\Sigma_i[\mathcal{N}])$.

2.2 Ehrenfeucht–Fraïssé Games

One of our main technical tools will be (various versions of) the *Ehrenfeucht–Fraïssé game*. In our context, the Ehrenfeucht–Fraïssé game for a set of numerical predicates, $\mathcal{N}$, is played by two players, Spoiler and Duplicator, on two strings $u, v \in A^*$. There is a fixed number k of rounds, and in each round i

- first, Spoiler chooses one position, a_i in u, or a position b_i in v;
- then Duplicator chooses a position in the other string, i.e., a b_i in v, if Spoiler's move was in u, and an a_i in u, otherwise.

After k rounds, the game finishes with positions $a_1, \ldots, a_k$ chosen in u and $b_1, \ldots, b_k$ chosen in v. Duplicator has won if the mapping $a_i \mapsto b_i$, $i = 1, \ldots, k$, is a *partial $\sigma_A \cup \mathcal{N}$–isomorphism*, i.e., if

- for every $i, j \leq k$, $a_i = a_j \iff b_i = b_j$,
- for every $i \leq k$, a_i and b_i carry the same letter, i.e., $u_{a_i} = v_{b_i}$, and
- for every m–ary predicate $P \in \mathcal{N}$, and every $i_1, \ldots, i_m \leq k$, it holds that $P(a_{i_1}, \ldots, a_{i_m}) \iff P(b_{i_1}, \ldots, b_{i_m})$.

If Duplicator has a winning strategy in the k–round game for $\mathcal{N}$ on two strings u and v, we write $u \equiv_k^{\mathcal{N}} v$. The fundamental use of the game comes from the fact that it characterises first–order logic (c.f., e.g., [EFT94]). In our context, this can be formulated as follows:

2.1 Theorem (Ehrenfeucht, Fraïssé)
A language $L \subseteq A^*$ is definable in $FO[\mathcal{N}]$ iff there is a finite subset $\mathcal{N}'$ of $\mathcal{N}$ and a number k such that, for every $u \in L, v \notin L$, Spoiler has a winning strategy in the k–round game for $\mathcal{N}'$ on u and v. □

We will also use the following variant of the game:
In the single–round k–game for $\mathcal{N}$ on two strings u, v

- first, Spoiler chooses k positions $a_1, \ldots, a_k$ in u, or $b_1, \ldots, b_k$ in v;
- then Duplicator chooses k positions in the other string, i.e., positions $b_1, \ldots, b_k$ in v, if Spoiler's move was in u, $a_1, \ldots, a_k$ in u, otherwise.

Again, Duplicator wins iff the mapping $a_i \mapsto b_i$, $i = 1, \ldots, k$, is a partial isomorphism. Clearly, if Duplicator has a winning strategy for the single–round k–game on u and v, then she also has one for the single–round h–game, for all $h \leq k$.

This game characterises the expressive power of $BC(\Sigma_1[\mathcal{N}])$:

2.2 Theorem
A language $L \subseteq A^*$ is definable in $BC(\Sigma_1[\mathcal{N}])$ iff there is a finite subset $\mathcal{N}'$ of $\mathcal{N}$ and a number k such that, for every $u \in L$, $v \notin L$, Spoiler has a winning strategy in the single–round k–game for $\mathcal{N}'$ on u and v. □

3 The Crane Beach Conjecture

Intuitively, since numerical predicates can only talk about *positions* in strings, it seems that they can only help express properties that depend on certain (combinations of) letters appearing in certain (combinations of) positions. The Crane Beach Conjecture (named after the location of its first, flawed, proof) is an attempt to make that intuition precise.

3.1 Definition (Neutral letter)
Let $L \subseteq A^*$. A letter $e \in A$ is called *neutral* for L if for any $u, v \in A^*$ it holds that $uv \in L \iff uev \in L$. □

Thus membership in a language with a neutral letter cannot depend on the individual positions on which letters are: any letter can be moved away from any position by insertion or deletion of neutral letters. It seems therefore conceivable that for every such language, if it can be defined at all in first–order logic then it can be defined using the linear order as the only numerical relation.

3.2 Definition (Crane Beach Conjecture)
Let $\mathcal{N}$ be a set of numerical predicates. We say that *the Crane Beach conjecture is true for* $\mathcal{N}$, iff every language $L \in FO[\leq, \mathcal{N}]$ that has a neutral letter is also definable in $FO[\leq]$. □

It turns out that the conjecture is true for some sets of numerical predicates, but not for all. In fact, it fails for the set $\mathcal{N} = \{+, *\}$. This set of predicates is particularly important because $FO[+, *]$ corresponds to the most natural uniform version of the circuit complexity class AC^0 [BIS90].
Our counterexample to the Crane Beach conjecture makes use of the well-known but somewhat counterintuitive ability of $FO[+, *]$ formulas to *count* letters up to numbers polylogarithmic in the input size:

3.3 Definition (Definibility of Counting)
Let $f(n) \leq n$ be a nondecreasing function from $\mathbb{N}$ to $\mathbb{N}$. We say that a logical system can *count up to* $f(n)$ if there is a formula φ such that for every n and for every $w \in \{0,1\}^n$,

$$w \models \varphi(c) \iff c \leq f(n) \wedge c = \#_1(w),$$

where $\#_1(w)$ is the number of ones in w.

We will need to consider two functions with similar notation. We write the base-two logarithm of n as $\lg n$, the k'th power of this logarithm as $(\lg n)^k$, and the k'th *iterated* logarithm as $\lg^{(k)} n$. For example, $\lg^{(2)} n$ is the same as $\lg(\lg n)$.

3.4 Proposition ([AB84, FKPS85, DGS86, WWY92])
The system $FO[+, *]$ can count up to $(\lg n)^k$ for any k. If $f(n) = (\lg n)^{\omega(1)}$, and $\mathcal{N}$ is any set of numerical predicates, then $FO[\leq, \mathcal{N}]$ cannot count up to $f(n)$.

3.5 Theorem
There is a language L with a neutral letter that is definable in $FO[+, *]$ but not in $FO[\leq]$.

Proof:

We define a language A on alphabet $\{0, 1, a\}$ as follows. For each positive integer k, A will contain a string consisting of the 2^k binary strings of length k, in order, separated by a's. The total length of the k'th string in A is thus $2^k(k+1) - 1$. The first three strings in A are thus $0a1$, $00a01a10a11$, and

$$000a001a010a011a100a101a110a111.$$

Our desired language B has alphabet $\{0, 1, a, e\}$ and is simply the set of strings w over this alphabet such that the string obtained by deleting all the e's in w is in A. Clearly B has a neutral letter e, as inserting or deleting e's cannot affect membership in B. Clearly B is not regular, so it cannot be in $FO[\leq]$. It remains for us to prove:

3.6 Lemma
B is definable in $FO[+, *]$.

Proof:
We need to formulate a sentence of $FO[+, *]$ that will hold for a string exactly if it is in B, that is, exactly if its non-neutral letters form a string in A. Recall that a string w is in A exactly if for some number k, w consists of the 2^w binary strings of length k, in order, separated by a's.
Our sentence will assert the existance of a number k such that the input string, with e's removed, is the k'th string in the language A. Since the length of the k'th string in A is exponential in k, and a valid input string must be at least as long, any valid k must be at most $\lg n$. Therefore by Proposition 3.4, the system $FO[+, *]$ is able to count letters in any interval in the input string up to a limit of k.
We first assert that there are exactly k 0's and no 1's before the first a, exactly k 0's and 1's between each pair of a's, exactly k 1's (and no 0's) after the last a. It then remains to assert that each string of 0's and 1's between two a's is the successor of the previous one. To do this, we assert that for every position y containing a 0 or 1:

- If there is a position w left of y such that there is a 0 or 1 at y and exactly $k-1$ 0's and 1's between w and y,
- Then w has the same letter as y *unless*
- x has the unique a between x and y, z has the next a to the right of x or is the rightmost position if there is no such a,
- w has 1, there are no 0's between w and x, y has 0, and there are no 1's between y and z, *or*
- w has 0, there are no 0's between w and x, y has 1, and there are no 0's between y and z.

This proves Lemma 3.6 and thus Theorem 3.5. □

Theorem 3.5 now follows immediately. □

The construction above crucially uses the fact that we can count up to $\lg n$ in $FO[+,*]$. We can strengthen the construction so that it provides a counterexample using only counting up to $\lg^{(m)} n$, the m times iterated logarithm of n. However, we do not yet know whether this strengthening is non-trivial — it may be that any set of numerical predicates that allows counting up to $\lg^{(m)} n$ also allows counting up to $\lg n$.

3.7 Proposition
If the system $FO[\leq,\mathcal{N}]$ can count up to $\lg^{(m)} n$ for some m, then there is a language L with a neutral letter that is definable in $FO[\leq,\mathcal{N}]$ but not in $FO[\leq]$.

Proof:
We must show that counting up to $\lg^{(m)} n$ suffices to provide a counterexample to the Crane Beach conjecture. We give the construction in some detail for $m = 2$, indicating how to generalize it to arbitrary values for m. Take the alphabet $\{a, b, 0, 1, e\}$ and for every k consider strings of the form $(b(0+1)^k(a(0+1)^k)^*)^*b$. Finally, add e as a neutral letter. a and b are used as markers, and we interpret the 0–1–substring between any two successive markers as the binary representation of some number between 0 and $2^k - 1$. If x is any position, we define $block(x)$ to be the interval between the two markers nearest x, and $num(x)$ to be the number represented by the 0–1 subsequence in $block(x)$. Using a formula that can count up to k and the construction from the proof of Theorem 3.5 we can write formulas expressing $num(x) = num(y)$ and $num(x) + 1 = num(y)$, respectively. We can now express easily that between every successive occurences of two b's each number from 0 to $2^k - 1$ is represented precisely once. In other words, this formula stipulates that the $\{a, 0, 1\}$–substring between two b's represent a permutation of the numbers $0, \ldots, 2^k - 1$. Finally, we write a formula that expresses that all permutations are represented. Altogether, our formula defines the set of those strings which consist of a sequence of permutations of the numbers $0, \ldots, 2^k - 1$, for some k, containing every permutation at least once. In particular, every such string has length $\Omega(2^k!)$, whereas counting is only required up to $k = O(\lg\lg(2^k!))$.
To be more precise, the formula forces all permutations to be present as follows. It says that for every represented permutation π (starting, say, with a b at position p), and every pair of positions i, j within that permutation (i.e., $p < i < j < p'$, where p' is the smallest position $> p$ that carries a b), there is a permutation ρ (between b's at q and q', say) which is equal to π, except that $num(i)$ and $num(j)$ are swapped. In what follows we will use abbreviations $first(x)$ and $last(x)$ for formulas which express that x lies in the first, respectively last, block of some permutation; $next(x)$ will denote the first position in the block directly to the right of $block(x)$. Our formula for i and j now expresses the following for all r, s such that $p < r < p'$ and $q < s < q'$:

- $num(r) = num(s) \rightarrow num(next(r)) = num(next(s))$
  unless $last(r)$ or $\{num(r), num(next(r))\} \cap \{num(i), num(j)\} \neq \emptyset$
- $(num(r){=}num(s) \wedge num(next(r)){=}num(i)) \rightarrow num(next(s)){=}num(j)$
- $(num(r){=}num(s) \wedge num(next(r)){=}num(j)) \rightarrow num(next(s)){=}num(i)$
- $(num(s){=}num(j) \wedge \neg last(s)) \rightarrow num(next(s)){=}num(next(i))$
- $(num(s) = num(i) \wedge \neg last(s)) \rightarrow num(next(s)) = num(next(j))$
- $(first(r) \wedge first(s) \wedge num(r) \neq num(i)) \rightarrow num(r) = num(s)$
- $(first(r) \wedge first(s) \wedge num(r) = num(i)) \rightarrow num(s) = num(j)$.

Thus we can construct the desired formula for $m = 2$.
We can then iterate this process, using an additional marker symbol c. The resulting formula stipulates that our string represent all permutations of all the permutations of the numbers $0, \ldots, 2^k - 1$. This will guarantee that string to be of length $\Omega(((2^k)!)!)$, etc. □

It is not difficult to code the languages above using only two non–neutral letters: just apply the homomorphism $\{a, b, 0, 1, e\}^* \rightarrow \{0, 1, e\}^*$ which maps e to e, a to 010, b to 0110, 0 to 01110, and 1 to 011110, for example. However, with only one non–neutral letter there is no way of defeating the conjecture.

3.8 Theorem
If $|A| = 2$ then for every set $\mathcal{N}$ of numerical predicates and every language $L \subseteq A^*$ with a neutral letter it holds that $L \in FO[\leq,\mathcal{N}] \Longrightarrow L \in FO[\leq]$.

Proof:
Let L be a language on $\{1, e\}$ with e as a neutral letter. Consider the set of numbers n such that 1^n is in L and 1^{n+1} is not. If this set is finite, it is easy to see that L is regular and definable in $FO[\leq]$. Otherwise, we will show that no family of unbounded fan-in circuits with constant depth and polynomial size can recognize L — it follows from [BIS90] that L is not definable in $FO[\leq,\mathcal{N}]$ for any $\mathcal{N}$.

For these particular values of n, any circuit deciding L on strings of length $2n$ would compute a symmetric function of the inputs saying yes on inputs with n 1's and no on inputs with $n+1$ ones. Following the construction of [FKPS85], a constant-depth poly-size combination of these circuits can be used to compute the parity function on inputs of this size. If the circuit deciding L had constant depth and polynomial size, then this new circuit would compute the parity function in AC^0 for infinitely many input sizes, violating [Ajt83, FSS84]. □

Since PARITY is a non–star–free regular language over $\{0,1\}^*$ and has a neutral letter, Theorem 3.8 implies the nonexpressibility of PARITY in first–order logic with arbitrary numerical predicates (i.e., AC^0). Note, however, that it directly uses the existing proofs of the nonexpressibility of PARITY to get this result.
On the other hand, the following special case of the Crane Beach conjecture can be proved directly:

3.9 Theorem
The Crane Beach conjecture holds for the set of all monadic relations.

Proof:
Let L be a language with a neutral letter that is not definable in $FO[\leq]$. This means that for any number of moves k there must be two strings $y \in L$ and $z \notin L$ such that the Duplicator wins the k-move game (using only $\leq$) on y and z. By adding neutral letters we can make y and z have the same length m.
Now let $\mathcal{N}$ be any monadic predicate. We will show that L is not definable in $FO[\leq, \mathcal{N}]$ as follows. We will use $\mathcal{N}$ to construct two strings $u \in L$ and $v \notin L$ from y and z by suitable padding with neutral letters. (The length of u and v will be a suitably large number n to be defined below.) Then we will show how the Duplicator can win the k-move game on u and v, with both $\leq$ and $\mathcal{N}$ as numerical predicates.
The predicate $\mathcal{N}$ may be regarded as a *coloring* of the input positions from 1 to n, with finitely many colors. If r and s are input positions, consider the colored string given by the interval from r to s, with each input position holding a neutral letter. For any two such strings, consider the k-move game with only $\leq$ as numerical predicate and the colors considered as the input. Let two strings be considered equivalent iff the Duplicator wins this game on them. Since the language defined by this game is regular, there are only a *finite number* of equivalence classes. We now define a colored undirected graph whose vertices are these n input positions and where the color of the edge from position r to position s represents the equivalence class of the colored string for that interval.
By the Erdos-Szekeres Theorem [ES35], as long as n is greater than m^d where d is the number of edge colors, there must be a *monochromatic path* in the graph of length at least m. We create u from y, and v from z, by placing the letters of the shorter strings in the locations given by the vertices of these path (the "special locations"), and making all other letters neutral. We must now explain how the Duplicator can win the game with $\leq$ and $\mathcal{N}$ on the strings u and v (the "Big Game").
The Duplicator will model the Big Game by a series of "small games", where she already has a winning strategy for each. One small game is played on the strings y and z using only $\leq$, and there is another small game (using $\leq$ and color only) for each interval between special locations. Whenever the Spoiler moves in the Big Game, the Duplicator translates this move into the y-z small game by moving to the position matching the next special position to the right. She also translates it into the small game for that interval. The Duplicator's reply in the Big Game is determined by her correct move in the y-z game, and her correct move in the special small game for that particular interval.
After k moves Delilah must win the original Small Game and all the interval Small Games, as she has made at most k moves in each. It is easy but tedious to look at the input predicates, order, equality, and position color in the Big Game and verify that Delilah has won that as well. □

We can use Theorem 3.9 to derive the following interesting generalization of the nonexpressibility of PARITY. But again, we do not get an *independent* proof of this fact because the existing proofs are used crucially to obtain the results in [BCST92].

3.10 Corollary
The Crane Beach conjecture holds for all regular languages. That is, for every set $\mathcal{N}$ of numerical predicates and every regular set L with a neutral letter it is true that that $L \in FO[\leq, \mathcal{N}] \Longrightarrow L \in FO[\leq]$.

Proof:
This follows from Theorem 3.9 and the fact, proven in [BCST92], that every regular language definable in $FO[\leq, \mathcal{N}]$ (using any set $\mathcal{N}$ of numerical predicates) is definable in $FO[\leq, \{mod_p \,/\, p \in \mathbb{N}\}]$, where $mod_p(i)$ is true iff $i \equiv 0 \mod p$. □

Although according to Theorem 3.9 the Crane Beach conjecture holds for the set of all unary relations, it is not true for all *binary* relations, since $FO[\leq, +, *] = FO[\leq, Bit]$, c.f., [Imm98]. In fact, it already fails for the set of all unary functions, or for the set of all linear orderings. This follows from the existence of a unary function $f : \mathbb{N} \to \mathbb{N}$ (see the proof of Theorem 3 in [Sch97]) and a set $\mathcal{O}$ of linear orderings (in fact, four order relations suffice, cf.[ScSc]) such that $FO[\leq, +, *] = FO[\leq, Bit] = FO[\leq, f] = FO[\leq, \mathcal{O}]$.

We can also consider special cases of the Crane Beach conjecture based on restrictions on the type of logical formulas allowed. For example, with arbitrary sets of numerical relations the conjecture does hold for Boolean combinations of Σ_1–formulas:

3.11 Theorem
Let $\mathcal{N}$ be a set of numerical predicates, and let L be a language with a neutral letter that is definable in the class $BC(\Sigma_1[\leq,\mathcal{N}])$. Then $L \in BC(\Sigma_1[\leq])$.

Proof:
We must show that for any set $\mathcal{N}$ of numerical predicates and any language L with a neutral letter, L is definable in $BC(\Sigma_1[\leq,\mathcal{N}])$ iff it is definable in $BC(\Sigma_1[\leq])$.
Using Theorem 2.2, we first show the proposition for the special case $\mathcal{N} = \{suc, \min, \max\}$, where suc is the successor relation $suc(n,m)$ iff $m = n{+}1$, $\langle w,n\rangle \models \min(n)$ iff $x{=}1$, and $\langle w,n\rangle \models \max(n)$ iff $n = |w|$.
Let e be the neutral letter, and assume that $L \notin BC(\Sigma_1[\leq])$. Then, for every k, there are strings $u \in L$, $v \notin L$ such that Duplicator wins the single–round k–game for $\leq$ on u, v. We can assume u and v to be of the same length m (if not, append $|v|{+}k$ e's to u and $|u|{+}k$ e's to v). We construct strings U from u and V from v such that $U \in L$, $V \notin L$, and Duplicator wins the single–round k–game for $\{\leq, suc, \min, \max\}$ on U, V. Then $L \notin BC(\Sigma_1[\leq, suc, \min, \max])$, which proves the assertion, by contraposition.
In order to construct U, insert $2k{-}1$ e's between each pair of adjacent positions in u, as well as at the beginning and the end of u. More precisely, $U = U_1 \cdots U_{m2k+2k-1}$, with $U_{j2k} = u_j$, and $U_{j2k+i} = e$, for any $j \leq m$, $i < 2k$. Similarly, we construct V from v. Since e is neutral, we have $U \in L, V \notin L$.
Assume that Spoiler chooses positions $a_1, \ldots, a_k$ in U (the other case is symmetric). Some (possibly all, or none) of the U_{a_j} will be neutral letters, others will be from $A \setminus \{e\}$. For the sake of notational simplicity we will assume, without loss of generality, that $U_{a_1}, \ldots, U_{a_q} \in A \setminus \{e\}$, and $U_{a_{q+1}} = \cdots = U_{a_k} = e$. Then each a_j with $j \leq q$ is of the form $s_j 2k$, for some $s_j \in \{1, \ldots, m\}$. Now Duplicator simulates a move of Spoiler in the game for $\leq$ on u, v in which Spoiler pebbles $s_1, \ldots, s_q$ on u, and finds her reply, $s'_1, \ldots, s'_q$ on v, according to her winning strategy. She then sets, for each j from 1 through q, b_j to be $s'_j 2k$. Then for each $j, j' \leq q$ it holds that

- $b_j \neq b_{j'}{+}1$ and $a_j \neq a_{j'}{+}1$,
- $b_j \leq b_{j'} \iff a_j \leq a_{j'}$, and
- $V_{b_j} = v_{s'_j} = u_{s_j} = U_{a_j}$.

To complete this move, Duplicator has to define $b_{q+1}, \ldots, b_k$ such that $V_{b_{q+1}} = \cdots = V_{b_k} = e$, and that for all $j, j' \leq k$

- $b_j \leq b_{j'} \iff a_j \leq a_{j'}$,
- $b_j = b_{j'}{+}1 \iff a_j = a_{j'}{+}1$, and
- $b_j = 1 \iff a_j = 1, b_j = |V| \iff a_j = |U|$.

Such $b_{q+1}, \ldots, b_k$ can easily be found, since between any two different b_i, b_j with $i, j \leq q$, there are at least $2k{-}1$ positions p where $V_p = e$.

Now let $\mathcal{N}$ be an arbitrary finite set of numerical predicates and assume that $L \notin BC(\Sigma_1[\leq])$. From what we have just shown it follows that, for every k, we can find strings $u \in L$, $v \notin L$ of the same length m such that Duplicator has a winning strategy in the single–round $2k{+}2$–game for $\leq, suc, \min, \max$ on u, v. We want to construct strings U and V by inserting neutral letters into u and v, respectively, in such a way that the original letters of u and v are moved onto positions $i_1, \ldots, i_m$ which are, in a certain sense, highly indistinguishable. To this end, we define, for every number n, a coloring of subsets of size $h \leq 2k$ of $\{1, \ldots, n\}$. This coloring was inspired by the one used by Straubing in [Str01], in his proof of Theorem 8. There he used the following extension of Ramsey's theorem, which will also help us here:

Theorem Let $m, k, c_1, \ldots, c_k > 0$, with $k \leq m$. Let n be sufficiently large as a function of m and the c's. If all h–element subsets of $\{1, \ldots, n\}$, with $1 \leq h \leq k$, are colored from a set of c_h colors, then there exists an m–element subset T of $\{1, \ldots, n\}$ such that for each h with $1 \leq h \leq k$ there exists a color κ_h such that all h–element subsets of T are colored κ_h. □

Let $\mathcal{T} = \{\tau_1, \ldots, \tau_q\}$ be the set of all atomic formulas over $\mathcal{N}, \leq$ on variables $x_1, \ldots, x_k, y_1, \ldots, y_h$. The *$\mathcal{N}, \leq$–type* of a tuple $r = (r_1, \ldots, r_k) \in \{1, \ldots, n\}^k$ with respect to a h–element set $S = \{p_1 < \cdots < p_h\}$, $\alpha(r, S)$, is the set of all those formulas of $\mathcal{T}$ that are satisfied when x_i is interpreted as r_i, and y_j as p_j, for $i \leq k$ and $j \leq h$.
We now color, for each number n and every $h \leq 2k$, every h–element set $S = \{p_1 < \cdots < p_h\} \subseteq \{1, \ldots, n\}$ with the set of all those $\alpha \subseteq \mathcal{T}$ for which there is a k–tuple r over $\{1, \ldots, n\}$ such that r has $\mathcal{N}$–type α with respect to S. Clearly, for every $h \leq 2k$ there is a fixed number of possible colors, independent of n. The extension of Ramsey's theorem stated above tells us that for large enough n we can find numbers $i_1 < \cdots < i_m \leq n$ such that, for every $h \leq 2k$, all h–element subsets of $\{i_1, \ldots, i_m\}$ have the same color. We now insert neutral letters into u in such a way that in the resulting string U we have $U_{i_s} = u_s$, for $s = 1, \ldots, m$, and $U_i = e$ for all $i \notin \{i_1, \ldots, i_m\}$. In the

same way we construct V from v. Let us call $i_1, \ldots, i_m$ the *special positions*.

We now show that Duplicator has a winning strategy in the k–game for $\leq, \mathcal{N}$ on U, V. Assume that Spoiler chooses $a = a_1, \ldots, a_k$ in U (again, the other case is symmetric). Then Duplicator finds, for every a_j the next smallest special position i_{s_j}, i.e, $i_{s_j} \leq a_j < i_{s_j+1}$. Let $S = \{i_{s_j}, i_{s_j+1} \,/\, j = 1, \ldots, k\}$. Duplicator now simulates a move of Spoiler in the $2k+2$–game for $\leq, suc, \min, \max$ on u, v, in which Spoiler plays all the points s_j and s_j+1, for $j = 1, \ldots, k$ on u, as well as min and max. Using her winning strategy in this game, Duplicator finds a reply with which she wins the game for $\leq, suc$. Therefore, we can safely call these points t_j, t_j+1, for $j = 1, \ldots, k$, and we know that $u_{s_j} = v_{t_j}$, for $j = 1, \ldots, k$. Let T be the set $\{i_{t_j}, i_{t_j+1} \,/\, j = 1, \ldots, k\}$. $|T| = |S| = h \leq 2k$, so S and T have the same colour, and this implies that there is a tuple $b = (b_1, \ldots, b_k)$ with the same $\mathcal{N}$–type as a, and with $\omega(b, T) = \omega(a, S)$. Duplicator now puts her pebbles on $b_1, \ldots, b_k$ in V. We have to check the winning conditions. By construction, $\alpha(a, S = \alpha(b, T)$. In particular, this implies that

- $(a_1, \ldots, a_k)$ and $(b_1, \ldots, b_k)$ have the same $\mathcal{N}$–type,
- $a_j \leq a_{j'} \iff b_j \leq b_{j'}$, for all j, j',
- if $a_j = i_{s_j}$ then $b_j = i_{t_j}$ hence $U_{a_j} = u_{s_j} = v_{t_j} = V_{b_j}$. If a_j is not of this form then $i_{s_j} < a_j < i_{s_j+1}$, consequently, $i_{t_j} < b_j < i_{t_j+1}$ and $U_{a_j} = V_{b_j} = e$.

□

As we have seen, with addition and multiplication first–order logic has enough expressive power to defeat the neutral letter. Addition alone is, in many ways much weaker than addition and multiplication together. For example, this is witnessed by the fact that the first–order theory of the natural numbers with $+$ and $*$ is undecidable, whereas Presburger arithmetic, the first–order theory of the natural numbers with addition only, can be decided using quantifier elimination. Also note that at least our technique for producing a counterexample cannot work with addition only, since it is well known (see, e.g., page 12 of [Lyn82]) that $FO[\leq, +]$ cannot count up to any non-constant function.

It is therefore more than conceivable that addition alone is too weak to make the conjecture fail, and we now show that this is indeed the case.

3.12 Theorem

Every language $L \in FO[\leq, +]$ that has a neutral letter is definable in $FO[\leq]$.

As indicated in the introduction, this theorem follows from collapse results for first–order queries over finite databases (e.g., Theorem 5.5 in [BST99]). However the terminology in which these results are formulated is rather alien to our setting here, so we will instead use a recent collapse result on *infinite* databases in [LS01]. First, however, let us give an intuitive explanation of the main idea behind the proof.

For simplicity, we concentrate on 0–1–strings u, v of the same (large) size and discuss what Duplicator has to do in order to win the k–round $+$–game on u and v. Let A be the set of indices a for which $u_a = 1$, similarly, $B = \{b \,/\, v_b = 1\}$. As in previous proofs, we will work with a set Q of indistinguishable positions, and choose u and v such that $A, B \subseteq Q$.

Assume that, after $i-1$ rounds $a^{(1)}, .., a^{(i-1)}$ have been played in u, and $b^{(1)}, .., b^{(i-1)}$ in v. Let Spoiler choose some element $a^{(i)}$ in u. When choosing $b^{(i)}$ in v, Duplicator has to make sure that any Spoiler moves for the remaining $k-i$ rounds in one structure can be matched in the other. In particular, this means that any sum over the $a^{(j)}$ behaves in relation to A exactly as the corresponding sum over the $b^{(j)}$ behaves in relation to B. For instance, for any sets $J, J' \subseteq \{1, .., i\}$, it should hold that there is some $a \in A$ that lies between $\sum_{j \in J} a^{(j)}$ and $\sum_{j' \in J'} a^{(j')}$ if and only if there is some $b \in B$ that lies between $\sum_{j \in J} b^{(j)}$ and $\sum_{j' \in J'} b^{(j')}$. But it is not enough to consider simple sums over previously played elements. Since with $O(r)$ additions it is possible to generate $s \cdot a^{(i)}$ from $a^{(i)}$, for any $s \leq 2^r$, we also have to consider linear combinations with coefficients as large as this. Furthermore, since Spoiler is allowed to choose either structure to move in each time, it is necessary to deal with even more complex linear combinations. One can only handle all these complications because, as the game progresses, the number of rounds left for Spoiler to do all these things decreases. This means, for instance, that the coefficients and the length of the linear combinations we have to consider decrease: after the last round, the only relevant linear combinations are simple additions of chosen elements.

All the technical details necessary to make this strategy work are worked out in [Lyn82] in order to prove that for each first–order formula with addition φ there is a set $Q \subseteq \mathbb{N}$ such that φ cannot distinguish between subsets of Q if they are of equal cardinality, or both large enough. Drawing on Lynch's theorem, in [LS01] the authors prove a theorem, which, specialised to our setting can be formulated as follows.

Theorem ([LS01], Theorem 3.2)

For every $k \in \mathbb{N}$ there exists a number $r(k) \in \mathbb{N}$ and an order–preserving mapping $q : \mathbb{N} \to \mathbb{N}$ such that, for every signature σ the following holds: If σ^U and σ^V are interpretations of σ over $\mathbb{N}$, and if $n, m \in \mathbb{N}$ with $\langle \mathbb{N}, \sigma^U, n \rangle \equiv^{\leq}_{r(k)} \langle \mathbb{N}, \sigma^V, m \rangle$, then $\langle \mathbb{N}, q(\sigma^U, n) \rangle \equiv^{+}_{k} \langle \mathbb{N}, q(\sigma^V, m) \rangle$. □

Here, $q(\sigma^U, n)$ is short for $\sigma^{q,U}, q(n)$, where $\sigma^{q,U} = \{R^{q,U} \,/\, R \in \sigma\}$, and $R^{q,U} = \{q(i) \,/\, i \in R^U\}$.

Proof of 3.12, using the above theorem:
Assume that $L \notin FO[\leq]$, and let $u = u_1 \cdots u_n \in L$, $v = v_1 \ldots v_m \notin L$, such that $u \equiv^{\leq}_{r(k)} v$. We construct strings $U \in L$, $V \notin L$ from u and v, respectively, by inserting neutral letters in such a way that $U_{q(i)} = u_i$ and $V_{q(j)} = v_j$, for $i = 1, \ldots, n$, $j = 1, \ldots, m$, where q is as in the theorem. u and v define σ_A–interpretations σ^U_A and σ^V_A, respectively, and the winning strategy of Duplicator on u and v can easily be extended to $\langle \mathbb{N}, \sigma^U, n\rangle$ and $\langle \mathbb{N}, \sigma^V, m\rangle$: If Spoiler plays a position $a_i \leq n$ on $\langle \mathbb{N}, \sigma^U, n\rangle$, this corresponds to a move on u, and Duplicator can choose her answer according to her winning strategy on v. If Spoiler plays a position $a_i > n$ on $\langle \mathbb{N}, \sigma^U, n\rangle$, then Duplicator replies with $b_i := m+(a_i-n)$. (The case where Spoiler plays on $\langle \mathbb{N}, \sigma^V, n\rangle$ is completely symmetric.) Clearly, this defines a winning strategy for Duplicator. Application of the theorem above gives us a winning strategy for Duplicator in the k round game for $\{\leq, +\}$ on $\langle \mathbb{N}, q(\sigma^U, n)\rangle$ and $\langle \mathbb{N}, q(\sigma^V, m)\rangle$. From this, we obtain a winning strategy for Duplicator in the k round game for $\{\leq, +\}$ on U and V: Every move of Spoiler in U is translated into a move on $\langle \mathbb{N}, q(\sigma^U, n)\rangle$, and Duplicator's reply on $\langle \mathbb{N}, q(\sigma^V, m)\rangle$ is translated back into a move on V. The winning condition of Duplicator on $\langle \mathbb{N}, q(\sigma^U, n)\rangle$ and $\langle \mathbb{N}, q(\sigma^V, m)\rangle$ directly translates into the winning condition for Duplicator on U and V, thus proving that $U \equiv^+_k V$. □

4 Discussion

Much of the above can be generalised from strings to arbitrary relational structures over the natural (or real) numbers. This programme is pursued in [LS01]. With regard to the questions here, the following problems remain open.

- It would be very good to have a proof of Theorem 3.8 that does not rely on [Ajt83, FSS84]. However, since Theorem 3.8 implies the nonexpressibility of PARITY, we expect this to be very difficult.
- What is the status of the conjecture for $FO[\leq, *]$? There is a construction of Julia Robinson [Rob49] defining addition from multiplication and the successor operation, but in our context this only suffices to define addition on some numbers (those less than $n^{1/4}$) from multiplication and order on *all* numbers. We conjecture that some variant of this construction will suffice to disprove the Crane Beach conjecture for $FO[\leq, *]$, perhaps by showing it equivalent to $FO[\leq, +, *]$.
- Can we find a set of numerical predicates that allows us to count up to $\lg^{(m)} n$, but not to $\lg n$? What about counting up to even smaller functions? We conjecture that the Crane Beach conjecture is true of a system iff it cannot count beyond a constant.
- Within $FO[\leq, +, *]$, we can consider the subclasses of formulas based on the number of quantifier alternations. The lg–counting operation requires Σ_3, and the construction of the counter example adds a few more levels. This leaves a gap between the upper bound of something like Σ_5 in Theorem 3.5, and a lower bound of $BC(\Sigma_1)$ in Theorem 3.11. Since in $BC(\Sigma_2)$, counting is only possible up to a constant (cf., [FKPS85]), it is conceivable that the lower bound can be improved.
- Theorem 3.12 places limits on the power of a particular uniform circuit complexity class, an "addition-uniform" version of AC^0. Can we use these techniques to place limits on the power of more powerful uniform versions of AC^0 (without using the non-uniform lower bounds) or on addition-uniform versions of more powerful classes? This has been done for one such class, an addition-uniform version of LOGCFL, by Lautemann, McKenzie, Schwentick, and Vollmer [LMSV99].
- It would also be of interest to study the conjecture for certain extensions of FO, such as FO with unary counting quantifiers or FO with modulo counting quantifiers. These each have various versions depending on the numerical predicates available.

References

[AB84] M. Ajtai and M. Ben-Or. A theorem on probabilistic constant depth computations. *Proc. 16th ACM STOC* (1983), 471-474.

[Ajt83] M. Ajtai. Σ^1_1–formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[BCST92] D.A.M. Barrington, K. Compton, H. Straubing, and D. Thérien. Regular languages in NC^1. *Journal of Computer and Systems Sciences*, 44:478–499, 1992.

[BIS90] D.A.M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comp. Syst. Sci.* **41:3** (1990), 274-306.

[BL00] M. Benedikt and L. Libkin. Expressive power: The finite case. In G. Kuper, L. Libkin, and J. Paredaens, editors, *Constraint Databases*, pages 55–87. Springer, 2000.

[BST99] O.V. Belegradek, A.P. Stolboushkin, and M.A. Taitslin. Extended order-generic queries. *Annals of Pure and Applied Logic*, 97:85–125, 1999.

[DGS86] L. Denenberg, Y. Gurevich, and S. Shelah. Definability by constant-depth polynomial-size circuits. *Inf. and Control* **70** (1986) 216-240.

[EFT94] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, New York, 2nd edition, 1994.

[ES35] P. Erdös and G. Szekeres. A combinatorial problem in geometry. *Compositio Math.* **2** (1935), 464-470.

[FKPS85] R. Fagin, M. Klawe, N. Pippenger, and L.J. Stockmeyer. Bounded-depth polynomial size circuits for symmetric functions. *Theoretical Computer Science*, 36:239–250, 1985.

[FSS84] M.L. Furst, J.B. Saxe, and M. Sipser. Parity, circuits, and the polynomial–time hierarchy. *Mathematical Systems Theory*, 17:13–27, 1984.

[Imm98] N. Immerman. *Descriptive and Computational Complexity*. Springer–Verlag, New York, 1998.

[LMSV99] C. Lautemann, P. McKenzie, T. Schwentick, and H. Vollmer. The descriptive complexity approach to LOGCFL. In *STACS 1999: 16th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **1563**, Springer Verlag (1999), 444-454.

[LS01] C. Lautemann and N. Schweikardt. An Ehrenfeucht–Fraïssé approach to collapse results for first–order queries over embedded databases. In *STACS 2001: 18th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **2010**, Springer Verlag (2001), 455-466.

[Lyn82] J. F. Lynch. On sets of relations definable by addition. *Journal of Symbolic Logic*, 47:659–668, 1982.

[Rob49] J. Robinson. Definability and decision problems in arithmetic. *Journal of Symbolic Logic* **14** (1949), 98-114.

[Sch97] Th. Schwentick. Padding and the expressive power of existential second-order logics. In Mogens Nielson and Wolfgang Thomas, editors, *Proceedings of the Annual Conference of the European Association for Computer Science Logic*, Lecture Notes in Computer Science, pages 461–477, 1997.

[ScSc] N. Schweikardt and Th. Schwentick. In preparation.

[Str94] H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhäuser, 1994.

[Str01] H. Straubing. Languages defined with modular counting quantifiers. *Information and Computation*, 2001. To appear.

[WWY92] I. Wegener, N. Wurm, and S.-Z. Yi. Symmetric functions in AC^0 can be computed in constant depth and very small size. in *Boolean Function Complexity*, ed. M. Paterson, *London Mathematical Society Lecture Notes* **169** (1992), 129-139.

"An $n!$ Lower Bound On Formula Size"

Micah Adler

Computer Science Dept.
UMass, Amherst, USA
http://www.cs.umass.edu/~micah

Neil Immerman*

Computer Science Dept.
UMass, Amherst, USA
http://www.cs.umass.edu/~immerman

Abstract

We introduce a new Ehrenfeucht-Fraïssé game for proving lower bounds on the size of first-order formulas. Up until now such games have only been used to prove bounds on the operator depth of formulas, not their size. We use this game to prove that the CTL$^+$ *formula* $\text{Occur}_n \equiv \mathbf{E}[\mathbf{F}p_1 \wedge \mathbf{F}p_2 \wedge \cdots \wedge \mathbf{F}p_n]$ *which says that there is a path along which the predicates* p_1 *through* p_n *occur in some order, requires size* $n!$ *to express in CTL. Our lower bound is optimal. It follows that the succinctness of* CTL$^+$ *with respect to* CTL *is exactly* $\Theta(n)!$. *Wilke had shown that the succinctness was at least exponential [Wil99].*

We also use our games to prove an optimal $\Omega(n)$ *lower bound on the number of boolean variables needed for a weak reachability logic* ($\mathcal{RL}^w$) *to polynomially embed the language* LTL. *The number of booleans needed for full reachability logic* $\mathcal{RL}$ *and the transitive closure logic* FO^2(TC) *remain open [IV97, AI00].*

1 Introduction

We introduce a new Ehrenfeucht-Fraïssé game for proving lower bounds on the size of first-order formulas. Previous such games only proved lower bounds on the quantifier depth of formulas.

We use this game to prove that the CTL$^+$ formula,

$$\text{Occur}_n \equiv \mathbf{E}[\mathbf{F}p_1 \wedge \mathbf{F}p_2 \wedge \cdots \wedge \mathbf{F}p_n] \qquad (1.1)$$

requires size $n!$ to express in CTL. The formula Occur_n says that there exists a path such that each of the predicates p_i occurs somewhere along this path. (**E** is the existential path quantifier: there exists a maximal path starting from the current point. **F** is the modal quantifier: somewhere now or in the future along the current path.)

This offers a quite different proof and improves the exponential lower bound on the succinctness of CTL compared with CTL$^+$ [Wil99]. We thus prove that the succinctness of CTL$^+$ with respect to CTL is exactly $\Theta(n)!$.

We prove that the parse tree of any CTL formula expressing Occur_n has at least $n!$ leaves. This bound is exactly optimal because the following formula expresses Occur_n and has $n!$ leaves in its parse tree. Here we use $[n]$ to denote $\{1, 2, \ldots, n\}$.

$$\varphi_n \equiv \bigvee_{i_1 \in [n]} \mathbf{EF}\Big(p_{i_1} \wedge \bigvee_{i_2 \in [n]-\{i_1\}} \mathbf{EF}\Big(p_{i_2} \wedge \bigvee_{i_3 \in [n]-\{i_1,i_2\}} \mathbf{EF}\big(\cdots \wedge \mathbf{EF}p_{i_n}\big)\cdots\Big)\Big)$$

The main contribution of these results is not so much the introduction of the new formula-size games, as their effective use proving a new and optimal result. Standard Ehrenfeucht-Fraïssé games are played on a single pair of structures $\mathcal{A}$, $\mathcal{B}$. They are used to prove lower bounds on the quantifier depth of a formula φ needed to distinguish $\mathcal{A}$ from $\mathcal{B}$. Our new game works on a whole set of structures A, B where all of A satisfies φ and all of B satisfies $\neg\varphi$. In a standard game, the pair of structures $\mathcal{A}$ and $\mathcal{B}$ may differ on a disjunction: $\varphi \equiv \alpha \vee \beta$. In this case they differ on α or they differ on β and the "or" may be discarded. However, in the formula- size game, the set of structures A must be split into two portions: A_1 satisfying α and A_2 satisfying β. All of B satisfies $\neg\alpha$ and $\neg\beta$. Thus the game on (A, B) is shifted to a pair of games, (A_1, B) and (A_2, B).

There are extensive connections between the computational complexity of a problem and its descriptive complexity, i.e., how complex a formula is needed to describe the problem. Descriptive complexity is measured via the size, number of variables, operator depth, etc. of the requisite formulas as a function of the size of the input structures being described [Imm99].

The formula-size games introduced here generalize standard EF games. They are also related to the communication complexity games that Karchmer and Wigderson used

*Research supported by NSF grant CCR-9877078.

0-7695-1281-X/01 $10.00 © 2001 IEEE

to prove lower bounds on the depth of monotone circuits [Kar89][1]. In the past, EF games have been useful in proving bounds on operator depth and number of variables, but they have not been used to prove lower bounds on the size of formulas. This has been a crucial lack, which the present paper takes a step in correcting.

The added complication of formula-size games means that we must build up considerable machinery to use them to prove lower bounds. Such lower bounds were heretofore unattainable for general first-order formulas. We believe that this game and the corresponding methods will have many applications.

In another application of formula-size games we show that $\Theta(n)$ booleans are needed to translate an LTL formula of size n to a polynomial-size formula of the reachability logic, $\mathcal{RL}^w$.

This paper is organized as follows: In §2 we provide the necessary background in logic including the introduction of transitive closure logic (FO(TC)) which provides the general setting for the games that we present. In §3 we review Ehrenfeucht-Fraïssé games and present the new formula-size games for FO(TC). In §4 we present the formula-size game for CTL. In §5 we define the graphs G_n over which we prove our lower bound. In §6 we prove our main result, the optimal $n!$ lower bound on the succinctness of CTL^+ with respect to CTL. In §7 we prove an $\Omega(n)$ lower bound on the number of boolean variables needed for $\mathcal{RL}^w$ to express Occur_n in polynomial size. In Appendix A we describe the language CTL and in Appendix B we describe reachability logic ($\mathcal{RL}$).

2 Background

In this section we review some basic definitions concerning finite model theory and transitive closure logic [Imm99].

The language $\mathcal{L}$ consists of first-order logic with unary relation symbols $\{p_n : n \in \mathbf{N}\}$, and binary relation symbol, R. By the *size* of a formula, we mean the number of nodes in its parse tree, i.e., the number of occurrences of logical connectives, quantifiers, operators, and atomic symbols.

For our purposes, a Kripke structure is a finite labeled graph:

$$\mathcal{K} \quad = \quad \langle S; p_n^{\mathcal{K}} : n \in \mathbf{N}; R^{\mathcal{K}} \rangle \tag{2.1}$$

where S is the set of states (vertices), each $p_n^{\mathcal{K}} \subseteq S$ is a unary relation on S, and $R^{\mathcal{K}} \subseteq S^2$ is the edge relation.

First-order logic $\mathcal{L}$ does not suffice to express such simple formulas as,

$$\text{"There is a path from where we are } (x) \text{ to a vertex where } p_{17} \text{ holds."} \tag{2.2}$$

For this reason we add a transitive closure operator to first-order logic to allow us to express reachability [Imm87].

Let the formula $\varphi(x_1, \ldots x_k, y_1, \ldots y_k)$ represent a binary relation on k-tuples. We express the reflexive, transitive closure of this relation using the transitive-closure operator (TC), as follows: $\text{TC}_{\overline{x},\overline{y}}\varphi$. Let FO(TC) be the closure of first-order logic under the transitive-closure operator. For example, the following formula expresses Equation 2.2: $(\exists y)[(\text{TC}_{x,y}R(x,y))(x,y) \wedge p_{17}(y)]$.

3 Ehrenfeucht-Fraïssé Games

We assume that the reader is somewhat familiar with classical Ehrenfeucht-Fraïssé (EF) games [Ehr61, Fra54, Imm99]. Typically there is a pair of structures $\mathcal{A}, \mathcal{B}$ and two players. Samson chooses vertices, trying to point out a difference between the two structures, and Delilah replies, trying to keep them looking the same. Typical games have a certain number of pebbles corresponding to variables, and rounds corresponding to the depth of nesting of quantifiers and other operators such as TC.

The typical fundamental theorem of EF games is that Delilah has a winning strategy for the k-pebble, m-move game on $\mathcal{A}, \mathcal{B}$ iff $\mathcal{A}$ and $\mathcal{B}$ agree on all k-variable, depth-m formulas. EF games are used to show nonexpressivity of a property Φ as follows: Delilah chooses a pair of structures $\mathcal{A}, \mathcal{B}$ that disagree on Φ but such that she has a winning strategy for the m-move, k-pebble game. It then follows that Φ is not expressible via a k-variable, depth-m formula.

We now present new games for proving lower bounds on formula size rather than depth. We first define the formula-size game for the language $\text{FO}^2(\text{TC})$ — first-order logic with 2 variables and the transitive closure operator. We chose this logic because it is simple, expressive, and quite general. It is easy to see how to generalize the game and its corresponding fundamental theorem to most reasonable logics by adding more variables and other operators. In the sequel we will specialize the $\text{FO}^2(\text{TC})$ game to a less general language, CTL, where we will prove our main results.

Definition 3.1 ($\text{FO}^2(\text{TC})$ Formula-Size Game) In the formula-size game, Delilah starts by picking two finite sets of structures: A_0, B_0. The root of the game tree is labeled A_0, B_0. (The intuitive idea is that there is some property Φ such that every structure in A_0 satisfies Φ ($A_0 \models \Phi$) and no structure in B_0 satisfies Φ ($B_0 \models \neg\Phi$).)

[1] Karchmer and Wigderson gave general games for proving lower bounds on circuit depth; but they proved lower bounds only using a monotone version of their games. They cast their games as a communication game in which two sets of structures differ on some property. Through successive bits of communication, each of which divides one of the sets of structures in half, eventually the sets are reduced to a collection of pairs where each pair differs on a particular bit. This is analogous to the closed nodes of our formula size game, in which each pair differs on a particular atomic formula.

At each move, Samson may play on any of the open leaves of the current game tree. (One of Samson's possible moves will be to close a leaf.) Suppose that the leaf that Samson chooses to play on is labeled with the pair of sets A, B.

"not" move: Samson switches the two sets letting the current leaf have a unique child labeled B, A.

"or" move: Samson splits A into two sets: $A = A' \cup A''$. He lets the current leaf have two children labeled A', B and A'', B.

$\exists$ move: Samson chooses a variable $v \in \{x, y\}$. He then assigns a value for v to every structure $\mathcal{A} \in A$. Delilah then answers by assigning a value for v to every structure $\mathcal{B} \in B$. Let A', B' be the two sets of structures with the new assignments for v. The current leaf is then given a child labeled A', B'.

TC move: Samson chooses a pair of previously assigned variables $v, v' \in \{x, y\}$. For every structure $\mathcal{A} \in A$, Samson then chooses a sequence of vertices from $\mathcal{A}$: $v^{\mathcal{A}} = a_0, a_1, a_2, \ldots, a_t = v'^{\mathcal{A}}$. Delilah then answers by choosing for every structure $\mathcal{B} \in B$ a similar sequence, $v^{\mathcal{B}} = b_0, b_1, b_2, \ldots, a_{t'} = v'^{\mathcal{B}}$. Samson then chooses a single consecutive pair b_i, b_{i+1} for each $\mathcal{B}$ and assigns x to b_i and y to b_{i+1}. The current leaf is then given a child labeled A', B' where B' is the result of these new assignments for each structure in B. A' consists of multiple copies of each structure $\mathcal{A} \in A$, one for each consecutive pair a_j, a_{j+1} in the sequence for $\mathcal{A}$ chosen by Samson and with x assigned to a_j and y assigned to a_{j+1}.

The idea behind this move is that Samson is asserting that every structure in A satisfies $\mathrm{TC}_{x,y}(\delta)(v, v')$ and no structure in B does. He thus presents what he claims is a δ-path from v to v' for each structure $\mathcal{A}$ in A. Delilah answers with a supposed δ path from v to v' for every $\mathcal{B}$ in B. Samson must then challenge one pair b_i, b_{i+1} in each of Delilah's supposed δ paths. He is in effect saying "$\neg\delta(b_i, b_{i+1})$". At the end of this move, every structure in A' should satisfy $\delta(x, y)$ and no structure in B' should.

atomic move: Samson chooses $v, v' \in \{x, y\}$ and an atomic formula $\alpha(v, v')$. (α can be $v = v'$, $R(v, v')$ or $p_i(v)$.) Samson can only make this move if every structure in A satisfies $\alpha(v, v')$ and no structure in B does. In this case, the current leaf is *closed.*

The object of the game for Samson is to close the whole game tree while keeping it as small as possible. Delilah on the other hand wants to make the tree as large as possible.

Delilah may make multiple copies of the structures in B before any of her moves. For this reason, there is an obvious strategy for Delilah that is optimal, namely do everything: in answer to an existential move, make a copy of $\mathcal{B}$ for each vertex in $\mathcal{B}$ and reply with that vertex. Similarly, in answer to a TC-move, Delilah can make enough copies of $\mathcal{B}$ and answer with all possible sequences without repetitions from v to v'. □

The reason that Delilah is allowed to make multiple copies in the size game is that otherwise Samson need not play relevant parts of the minimal formula separating A and B. For example, suppose that $A = \{\mathcal{A}\}$ and $B = \{\mathcal{B}\}$ each consist of a single structure. Suppose that the smallest formula true of $\mathcal{A}$ but not $\mathcal{B}$ is,

$$\exists x \exists y (p_1(x) \leftrightarrow p_1(y) \ \wedge \ p_2(x) \leftrightarrow p_2(y) \wedge \cdots \wedge \ p_n(x) \leftrightarrow p_n(y)),$$

i.e., $\mathcal{A}$ has two points agreeing on all n predicate symbols, but $\mathcal{B}$ does not. If Delilah could not make duplicates, then Samson could just choose the relevant x and y in $\mathcal{A}$ and Delilah would have to answer with a single pair from $\mathcal{B}$. Then either the x's or the y's would differ on some predicate symbol p_i and Samson could close a game tree of size 3, rather than n.

The fundamental theorem of the formula-size game is:

Theorem 3.2 *Samson can close the game started at A_0, B_0 in a tree of size s iff there is a formula $\varphi \in \mathrm{FO}^2(\mathrm{TC})$ of size at most s such that every structure in A_0 satisfies φ and no structure in B_0 does.*

Proof: Suppose that φ of size s separates A_0 and B_0. Then Samson can "play φ" and a closed game tree of size s will result. Playing φ means the following. Suppose that $A \models \varphi$ and $B \models \neg\varphi$.

$\varphi = \neg\psi$: Samson plays the "not" move. In the resulting leaf $A' \models \psi$ and $B' \models \neg\psi$.

$\varphi = \psi \vee \rho$: Samson plays the "or" move letting A' be the subset of A satisfying ψ, and A'' the subset satisfying ρ. Thus one child differs on ψ and the other differs on ρ.

$\varphi = (\exists v)\psi$: Samson plays the $\exists$ move assigning v to a witness for ψ in every structure of A. Thus, whatever Delilah answers we have that $A' \models \psi$ and $B' \models \neg\psi$.

$\varphi = \mathrm{TC}_{x,y}(\delta)(v, v')$: Samson plays the TC move and as argued in the discussion after the definition of this move, $A' \models \delta$ and $B' \models \neg\delta$.

φ is atomic: Samson plays the atomic move, using φ and succeeds in closing this leaf.

Conversely, suppose that Samson has succeeded in closing the game in size s and that Delilah has played optimally. It follows that the resulting game tree is a size s formula satisfied by all of A_0 and none of B_0.

This can be seen inductively from the leaves of the closed game tree. For closed leaf, (A, B), $A \models \alpha$ and $B \models \neg\alpha$, where α is an atomic formula, i.e., has size one.

Inductively, assume that (A, B) has children (A_i, B_i) each differing on a formula ψ_i of size s_i where $i = 1$ for "not", $\exists$ and TC moves and $i = 1, 2$ for the "or" move. Here s_i is the size of the subtree rooted at (A_i, B_i).

"not" move: $A \models \neg\psi_1$, $B \models \psi_1$ and thus they differ on a formula of size $s_1 + 1$.

"or" move: $A \models \psi_1 \vee \psi_2$, $B \models \neg(\psi_1 \vee \psi_2)$ and thus they differ on a formula of size $s_1 + s_2 + 1$.

$\exists$ move: $A \models (\exists v)\alpha_1$, $B \models \neg(\exists v)\alpha_1$, and thus they differ on a formula of size $s_1 + 1$. Note that since Delilah plays optimally, if it were the case that some $\mathcal{B} \in B$ satisfies $(\exists v)\alpha_1$, then Delilah would have chosen the appropriate witness for this $\mathcal{B}$ and it would not have been the case that $B_1 \models \neg\alpha_1$.

TC move: $A \models \mathrm{TC}_{x,y}(\alpha_1)(v, v')$, $B \models \neg\mathrm{TC}_{x,y}(\alpha_1)(v, v')$, and thus they differ on a formula of size $s_1 + 1$. By the definition of the TC move, since $A_1 \models \alpha_1$, we know that for every $\mathcal{A} \in A$, there is an α_1-path from $v^{\mathcal{A}}$ to $v'^{\mathcal{A}}$. Furthermore, if there were an α_1-path from $v^{\mathcal{B}}$ to $v'^{\mathcal{B}}$, for some $\mathcal{B} \in B$, then Delilah would have played it for one of her copies of $\mathcal{B}$. Therefore, no matter which consecutive pair in this path Samson challenged, it would satisfy α_1.

Thus A_0 and B_0 differ on a formula of size s. □

4 Definition of the CTL Game

For a definition of CTL see the appendix or [CGP99]. We now define the CTL formula-size game[2]. This is a restriction of the $\mathrm{FO}^2(\mathrm{TC})$ formula-size game (Definition 3.1) as follows.

- There is only a single pebble name: x.
- The "not" and "or" moves are unchanged.
- The atomic move is unchanged except that it is played only using atomic formulas p_i.
- The $\exists$ and TC moves are replaced by the following, played on a leaf, ℓ, labeled with the pair of sets A, B,

  EX move: For each $\mathcal{A} \in A$, Samson reassigns x to a child of the current x. Delilah answers by first making as many copies of each $\mathcal{B} \in B$ as she wishes. For each copy $\mathcal{B} \in B$ she assigns x to a child of the current x. The resulting node labeled A', B' becomes the only child of ℓ.

  EU move: For each $\mathcal{A} \in A$, Samson chooses a path of length zero or more: $x^{\mathcal{A}} = a_0, a_1, \ldots, a_r$. Delilah answers as above with a path $x^{\mathcal{B}} = b_0, b_1, \ldots, b_s$ for each copy she makes of each $\mathcal{B} \in B$. Samson is trying to assert that $(A, x) \models \mathbf{E}(\alpha \mathbf{U} \beta)$, i.e., that $(A, a_i) \models \alpha$ for $i < r$, and $(A, a_r) \models \beta$. Presumably this holds for all of Samson's chosen paths and none of Delilah's.

  In the second half of the move, Samson divides the paths chosen by Delilah into two sets. For the first set he assigns x to some b_i with $i < s$ and puts these structures into B_1. For the second set he assigns x to b_s and puts these structures into B_2. Delilah answers by making enough copies so that she can assign x to each possible point in Samson's paths. When she assigns x to the final point b_r in a path, she puts that structure in A_2. When she assigns x to a non-final point she puts that structure into A_1. The node ℓ now has two children labeled A_1, B_1 and A_2, B_2 respectively.

  Intuitively what has happened in the second half of this move is that for those paths chosen by Delilah whose final points do not satisfy β, Samson chooses this point and puts the structure into B_2. For those paths one of whose non-final points does not satisfy α, Samson chooses this point and puts the structure into B_1. At the end of the move we have that $A_1 \models \alpha$, $B_1 \models \neg\alpha$, $A_2 \models \beta$, and $B_2 \models \neg\beta$. If the set B_1 or B_2 should happen to be empty then that node is considered closed.

  AU move: This is similar to the **EU** move except that the first half of the move now has two parts: (a) Samson chooses a maximal path for each structure in B, and Delilah makes copies and chooses a maximal path for each copy of each structure in A; (b) Samson chooses a finite initial segment of each path chosen by Delilah and then Delilah chooses a finite initial segment of each path chosen by Samson. Delilah may make copies of the paths chosen by Samson in order to choose more than one initial segment from each path. The second half of the move is the same as for the **EU** move.

It should be clear from the above definition and the proof of Theorem 3.2 that the following theorem holds:

[2] It is easy to generalize this also to the CTL* formula-size game, but we leave this to the reader.

Theorem 4.1 *Samson can close the CTL formula-size game started at A_0, B_0 in a tree of size s iff there is a formula $\varphi \in$ CTL of size at most s such that every structure in A_0 satisfies φ and no structure in B_0 does.*

5 Setting Up the Playing Field

In this section we describe the graphs on which we will play the CTL game to prove our main lower bound, Theorem 6.1. For each $n > 1$, we will build two sets of graphs A_0, B_0 such that $A_0 \models \text{Occur}_n$ and $B_0 \models \neg\text{Occur}_n$. For each of the $n!$ possible paths that might satisfy Occur_n, A_0 will include one graph that contains this path. Furthermore, we give each graph in A_0 and B_0 copies of all permutations of length $n-1$. This will help make A_0 and B_0 very difficult to distinguish.

For any fixed $n > 1$ consider the following directed graph, $G_n = (V_n, E_n)$. Let $\Pi_{[n]}$ be the set of all permutations π on any nonempty subset of $[n]$ and let Π_n be the set of permutations on the full set $[n]$. The vertices of G_n consist of the union of two sets, $V_n = T_n \cup F_n$,

$$T_n = \{t_\pi \mid \pi \in \Pi_{[n]}\}; \quad F_n = \{f_\pi \mid \pi \in \Pi_{[n]}\}$$

We represent the permutation $\pi \in \Pi_{[n]}$ as a 1:1 map,

$$\pi : [|\text{rng}(\pi)|] \rightarrow \text{rng}(\pi) \subseteq [n] .$$

For any such permutation π on at least two elements, define its tail, $\text{tail}(\pi) : [|\text{rng}(\pi)| - 1] \rightarrow \text{rng}(\pi) - \{\pi(1)\}$ where $\text{tail}(\pi)(i) = \pi(i+1)$. For ease of notation, let $\pi^2 = \text{tail}(\pi)$, and in general, $\pi^{k+1} = \text{tail}^k(\pi)$, i.e., the permutation π starting from item $k+1$.

For all $\pi \in \Pi_{[n]}$, the relation $p_{\pi(1)}$ holds of vertex t_π. Also, if π is a permutation on at least two elements then $p_{\pi(1)}$ holds of vertex f_π.

The node f_π has edges to the following successors nodes:

- $t_\sigma \in T_n$ where $\text{rng}(\sigma) \subseteq \text{rng}(\pi) - \{j\}$, for some $j \in \text{rng}(\pi), j \neq \pi(1)$
- $f_\sigma \in F_n$ where $\text{rng}(\sigma) \subseteq \text{rng}(\pi) - \{j\}$, for some $j \in \text{rng}(\pi)$

The node t_π has edges to all the successors of f_π together with the additional successor t_{π^2}. Furthermore, every vertex in V_n has an edge back to itself.

Consider the following sets of vertices and structures,

$$\begin{aligned} Y_n &= \{t_\pi \in T_n \mid \pi \in \Pi_n\} \\ N_n &= \{f_\pi \in F_n \mid \pi \in \Pi_n\} \\ A_0 &= \{(G_n, t_\pi) \mid t_\pi \in Y_n\} \\ B_0 &= \{(G_n, f_\pi) \mid f_\pi \in N_n\} \end{aligned}$$

The idea behind G_n is that for each $\pi \in \Pi_n$, t_π and f_π are very difficult to distinguish. However, observe that,

Lemma 5.1 *For any $\pi \in \Pi_n$,*

$$(G_n, t_\pi) \models \text{Occur}_n; \quad \textit{but} \quad (G_n, f_\pi) \models \neg\text{Occur}_n$$

6 Playing the CTL Game

In this section we prove the following,

Theorem 6.1 *The formula Occur_n (Equation 1.1) cannot be expressed in a CTL formula of size less than $n!$. Thus, there is a CTL$^+$ formula of size $O(n)$ whose smallest equivalent CTL formula has size $n!$.*

Corollary 6.2 *The succinctness of CTL$^+$ with respect to CTL is exactly $\Theta(n)!$.*[3]

By Lemma 5.1 we have that $A_0 \models \text{Occur}_n$ and $B_0 \models \neg\text{Occur}_n$. To prove Theorem 6.1 it suffices to show the following,

Lemma 6.3 *Samson cannot close the CTL-game on (A_0, B_0) in a game tree with fewer than $n!$ leaves.*

We will prove Lemma 6.3 through a series of additional lemmas. Since there is only one structure namely G_n on which we are playing and the only thing that matters is where x is assigned, we will abbreviate the structure $\mathcal{A}$ for which $x^{\mathcal{A}} = a$ by the point a. Thus a tree node will be labeled A, B with A and B boths sets of vertices from G_n.

We say that a pair $\langle a, b\rangle$ *occurs* at a node v of a game tree if v is labeled (A, B) and $a \in A$, $b \in B$. The following lemma is obvious but useful:

Lemma 6.4 *If a pair $\langle a, a\rangle$ occurs anywhere in a game tree, then that tree can never be closed.*

Let $\mathcal{T}$ be a closed game tree whose root is labeled (Y_n, N_n) and on which Delilah and Samson have both played perfectly. We will argue that $\mathcal{T}$ has at least $n!$ leaves.

Lemma 6.5 *Let $\pi \in \Pi_n$. Then there is a branch in $\mathcal{T}$ from root to leaf along which the following pairs occur (in this order),*

$$\langle t_\pi, f_\pi\rangle, \langle t_{\pi^2}, f_{\pi^2}\rangle, \langle t_{\pi^3}, f_{\pi^3}\rangle, \ldots, \langle t_{\pi^n}, f_{\pi^n}\rangle$$

Proof: By definition of Y_n, N_n, $\langle t_\pi, f_\pi\rangle$ occurs at the root. Suppose inductively that $\langle t_{\pi^k}, f_{\pi^k}\rangle$ occurs at node v_k (and is preceded by $\langle t_{\pi^j}, f_{\pi^j}\rangle$ for all $j < k$); and v_k is the lowest node at which $\langle t_{\pi^k}, f_{\pi^k}\rangle$ occurs. If $k = n$, then the lemma is proved. Suppose that $k < n$. In this case, v_k is an

[3] See Emerson and Halpern [EH85] for the upper bound.

open node since t_{π^k} and f_{π^k} both satisfy the same predicate symbol, $p_{\pi(k)}$.

From now on, let us assume that there are no "not" moves, but that instead Samson may play on the left or on the right. This may slightly decrease the size of $\mathcal{T}$ by removing "not" moves, but the number of leaves is unchanged. Note that an "or" move on the right is really an "and" move, and an **E** move on the right is really an **A** move.

Observe that if Samson plays an "or" move at v_k, then the pair $\langle t_{\pi^k}, f_{\pi^k}\rangle$ would still occur at one of v_k's children. Furthermore, Samson cannot close v_k. Thus, Samson must play one of the following moves: **EX**, **EU**, **AU**.

Recall that every path from f_π is also a path from t_π. Thus if Samson plays on the right, stepping off f_π to some descendant d, then t_π has the identical descendant d which Delilah will play. It follows from Lemma 6.4 that, Samson must play on the left at v_k.

If Samson plays **EX** then he must move from t_{π^k} to one of its successors. The only successor of t_{π^k} that is not a successor of f_{π^k} is $t_{\pi^{k+1}}$. Thus, Samson must move to $t_{\pi^{k+1}}$ and Delilah will move to all successors of f_{π^k}, including $f_{\pi^{k+1}}$. Thus $\langle t_{\pi^{k+1}}, f_{\pi^{k+1}}\rangle$ occurs in the child of v_k as desired.

Suppose that Samson plays **AU**. Samson starts by choosing a maximal path for each structure on the left. Delilah answers by choosing the infinite loop on the current vertex for each structure on the right. Recall that G_n has a self-loop at each vertex. Now, Samson chooses an initial segment of each infinite self-loop. Delilah responds by choosing the initial segments of length zero from Samson's paths. The right child of v_k is thus labeled exactly the same as v_k. Thus it is not useful for Samson to play **AU**.

Finally, suppose that Samson plays **EU**. He chooses a path from t_{π^k} to some descendant d. Note that if $d \neq t_{\pi^{k+1}}$ then d is also a descendant of f_{π^k}. Thus Delilah will answer with the path consisting of a single step from f_{π^k} to d. If Samson challenges f_{π^k} then we have made no progress. If Samson challenges d, then the right child of v_k contains the pair $\langle d, d\rangle$ and thus Delilah wins. Thus, Samson must play the path from t_{π^k} to $t_{\pi^{k+1}}$. Delilah will answer among others with the path from f_{π^k} to $f_{\pi^{k+1}}$ and $\langle t_{\pi^{k+1}}, f_{\pi^{k+1}}\rangle$ occurs at a child of v_k as desired. □

The path of permutation π which is guaranteed by Lemma 6.5 to occur along at least one branch of $\mathcal{T}$ may in fact occur along several branches. For each permutation π we would like to choose a particular branch as the representative branch of π. If $\langle t_{\pi^k}, f_{\pi^k}\rangle$ occurs at v along this branch, and $\langle t_{\pi^k}, f_{\pi^k}\rangle$ still occurs at one of v's children, then we follow this child, i.e., we take a branch that avoids making progress if possible. If both steps make progress, or neither do, we follow the left child.

Let π, σ be distinct elements of Π_n. In the next lemma we prove that the branches of π and σ must diverge at some point in $\mathcal{T}$. By this we mean that the branches start together at the root, but eventually separate and end at distinct leaves. It will then follow that there are at least as many leaves of $\mathcal{T}$ as elements of Y_n and Lemma 6.3 and Theorem 6.1 thus follow.

Lemma 6.6 *Let π, σ be distinct elements of Π_n. Then the branches of π and σ diverge.*

Proof: Let us assume for the sake of a contradiction that the branches of π and σ coincide. Let k be the first place that π and σ differ, i.e., $\pi(i) = \sigma(i)$ for $i < k$ and $\pi(k) \neq \sigma(k)$. We know that $\langle t_\pi, f_\pi\rangle$ and $\langle t_\sigma, f_\sigma\rangle$ both occur at the root.

The branches for π and σ may be moving down in lock step, i.e., $\langle t_{\pi^i}, f_{\pi^i}\rangle$ occurs at the same node as $\langle t_{\sigma^i}, f_{\sigma^i}\rangle$ or one may be ahead of the other, e.g., $\langle t_{\pi^{i+1}}, f_{\pi^{i+1}}\rangle$ occurs at the same node as $\langle t_{\sigma^i}, f_{\sigma^i}\rangle$. Let us assume that they are in lock step, or π is ahead of σ when $\langle t_{\sigma^{k+1}}, f_{\sigma^{k+1}}\rangle$ first occurs. Let v_k be the lowest node on the branch at which $\langle t_{\sigma^k}, f_{\sigma^k}\rangle$ occurs. Since $\langle t_{\sigma^k}, f_{\sigma^k}\rangle$ does not occur as a child of v_k, Sampson must play either **EX** or **EU** at the node v_k. There are two cases.

Case 1: $\langle t_{\pi^k}, f_{\pi^k}\rangle$ also occurs at v_k. Thus Samson must step from t_{π^k} to $t_{\pi^{k+1}}$ and from t_{σ^k} to $t_{\sigma^{k+1}}$ at this step. Since $\pi(k) \neq \sigma(k)$, $t_{\pi^{k+1}}$ is a descendant of f_{σ^k} (and $t_{\sigma^{k+1}}$ is a descendant of f_{π^k}). If Samson challenges either of these descendants, then we have the same point on both sides of a node in $\mathcal{T}$ and Delilah wins. If Samson challenges neither, then $\langle t_{\sigma^k}, f_{\sigma^k}\rangle$ occurs at a proper descendent of v_k, contradicting our assumption.

Case 2: $\langle t_{\pi^j}, f_{\pi^j}\rangle$ occurs at v_k for $j > k$. Samson must step from t_{σ^k} to $t_{\sigma^{k+1}}$ and either leave t_{π^j} fixed, or step from t_{π^j} to $t_{\pi^{j+1}}$. Let d be the not-necessarily-proper descendant of t_{π^j} that Samson steps to. Delilah answers with the path from f_{σ^k} to d. Since we have assumed that progress on σ is made at this node, Samson cannot challenge f_{σ^k}. Thus he must challenge d and the pair $\langle d, d\rangle$ occurs at the left child of v_k. This contradicts our assumption that $\mathcal{T}$ is closed.

Thus we have proved that the branches of π and σ cannot remain together after the second one has moved past level k. □

7 Lower Bound on Booleans in Reachability Logic

In this section we give an interesting application of formula-size games to characterize the number of boolean variables needed in a reachability logic. In [IV97] it is shown that CTL* is linearly embedable in the transitive closure logic $FO^2(TC)$. Furthermore in [AI00] a sublanguage of $FO^2(TC)$ called reachability logic ($\mathcal{RL}$) is described.

CTL* remains linearly embedable in $\mathcal{RL}$. The complexity of checking whether a Kripke structure, $\mathcal{K}$, satisfies an $\mathcal{RL}$ formula, φ, is $O(|\mathcal{K}||\varphi|2^{n_b})$ where n_b is the number of boolean variables occurring in $\mathcal{RL}$. Both $\mathcal{RL}$ and FO$^2(TC)$ may contain boolean-valued variables in addition to their two domain variables. Since the time to model check is linear in the size of the formula and the size of the structure, but exponential in the number of booleans, information about how many booleans are needed is important.

The boolean variables are not needed to embed CTL; however in the linear embeddings of CTL* in $\mathcal{RL}$ and FO^2(TC) at most a linear number of boolean variables may be used. It was left open in [IV97] whether any such booleans are actually needed. It was shown in [AI00] that at least one boolean is needed to embed CTL* at all in FO^2(TC) or $\mathcal{RL}$. Whether more than one such boolean variable is needed remains open.

In this section we use a size game for a weakened version of $\mathcal{RL}$ which we call $\mathcal{RL}^w$. The main result of this section is that for the formulas Occur$_n$ to be translated to polynomial-size formulas in $\mathcal{RL}^w$, $\Theta(n)$ boolean variables are needed. The main weakness of $\mathcal{RL}^w$ is that we do not allow new unary relations to be defined. We also require weak adjacency formulas to imply $R(x,y)$ as opposed to $R(x,y) \vee R(y,x) \vee x = y$, but this is just for convenience. It can be shown that LTL $\subseteq \mathcal{RL}^w$ but CTL $\not\subseteq \mathcal{RL}^w$. Due to lack of space we do not give a full explanation of $\mathcal{RL}$, directing the reader instead to [AI00]. (We do provide the definition of $\mathcal{RL}$ and a few examples in Appendix B.)

Our original motivation in trying to prove lower bounds on the formula Occur$_n$ was to characterize how many boolean variables are needed in the translations of CTL* to FO^2(TC) and $\mathcal{RL}$. In this section we are only able to prove a good lower bound for translations to the weaker language $\mathcal{RL}^w$. We believe that even this partial result is of interest, and we suspect this approach will lead to a similar lower bound for the full $\mathcal{RL}$.

Definition 7.1 A *weak adjacency formula* $\delta(x, \bar{b}, y, \bar{b'})$ is the conjunction of $R(x,y)$ with a boolean combination of the booleans $\bar{b}$, $\bar{b'}$ and the unary relations $p_i(x)$, $p_i(y)$. Define $\mathcal{RL}^w$ to be the smallest fragment of FO^2(TC) that satisfies the following:

1. If p is a unary relation symbol then $p \in \mathcal{RL}^w$.
2. If $\varphi, \psi \in \mathcal{RL}^w$, then $\neg\varphi \in \mathcal{RL}^w$ and $\varphi \wedge \psi \in \mathcal{RL}^w$.
3. If $\varphi \in \mathcal{RL}^w$ and $\delta(x, \bar{b}, y, \bar{b'})$ is a weak adjacency formula then the following formulas are in $\mathcal{RL}^w$:
   (a) REACH$(\delta)\varphi$
   (b) CYCLE(δ)

Semantics of $\mathcal{RL}^w$:

$$\begin{aligned} p &\equiv p(x) \\ \text{REACH}(\delta)\varphi &\equiv \exists y(\text{TC}\,\delta)(x, \bar{0}, y, \bar{1}) \wedge \varphi[y/x]) \\ \text{CYCLE}(\delta) &\equiv (\text{TC}\,\delta)(x, \bar{0}, x, \bar{1}) \end{aligned}$$

□

As an example, we translate Occur$_n$ to $\mathcal{RL}^w$ as follows: Occur$_n \equiv$ REACH(δ_n)**true** where $\delta_n(x, \bar{b}, y, \bar{b'}) \equiv R(x,y) \wedge \bigwedge_{i=1}^{n}(b'_i \rightarrow (b_i \vee p_i(x)))$.

The idea is that boolean variable b_i keeps track of whether predicate p_i has ever been satisfied in the current path. We can reach a point where all the booleans are one iff Occur$_n$ holds.

The $\mathcal{RL}^w$ formula-size game is very similar to the CTL formula-size game. In the Reach move, Samson asserts that REACH$(\delta)\varphi$ holds for all the vertices $v_0 \in A$. For each such v_0 he produces a path:

$$(v_0, \overline{b^0}), (v_1, \overline{b^1}), \ldots, (v_r, \overline{b^r})$$

where $\overline{b^0} = \bar{0}$, $\overline{b^r} = \bar{1}$, and $R(v_i, v_{i+1})$ holds for all $i < r$. Delilah answers with a similar path,

$$(w_0, \bar{0}), (w_1, \overline{c^1}), \ldots, (w_s, \bar{1}),$$

for as many copies as she wishes of each $w_0 \in B$. For each of Delilah's paths, Samson either challenges the final point, w_s, and puts it in B_2, or he challenges some pair $\langle (w_i, \overline{c^i}), (w_{i+1}, \overline{c^{i+1}}) \rangle$ and puts it in B_1. Then Delilah lets A_2 contain all the v_r's and A_1 contains all pairs, $\langle (v_i, \overline{b^i}), (v_{i+1}, \overline{b^{i+1}}) \rangle$. If originally A and B differed on REACH$(\delta)\varphi$ then after the move, A_1 and B_1 differ on δ and A_2 and B_2 differ on φ. Note that δ is quantifier free and only concerns the booleans together with the unary predicates true at the two points of each pair. In the game we consider below Delilah will only play pairs that correspond to pairs played by Samson, so Samson will never challenge a pair, but rather the endpoint of each of Delilah's paths.

The Cycle move is similar to the Reach move. Since the graphs we will play on below are acyclic, it will not be useful for Samson to play the Cycle move. Let the $\mathcal{RL}^w_k$ game be the $\mathcal{RL}^w$ game in which the tuples of booleans are of size at most k.

We next define the graph H_n on which we will play the $\mathcal{RL}^w$ game. These are simpler than the G_n from Section 5 because we only need an exponential lower bound, not an $n!$ lower bound. Thus we only need consider all subsets of the n propositional variables, not all possible paths through them.

Let X_n be the set of all proper subsets of the n predicates. For any element e of X_n, let $S(e)$ be a path that visits every predicate of e exactly once, and then visits a blank vertex. Let $F(e)$ be a path that visits every predicate

of e exactly once. The order of the predicates in $F(e)$ and $S(e)$ does not matter.

H_n contains $2^n - 1$ "true" vertices, t_e, one for each $e \in X_n$. Node t_e starts with the path $S(e)$, and then from the last (blank) vertex — call it b_e — there is an edge to each first vertex of $F(f)$, for any $f \in X_n$ such that $e \cup f \neq [n]$ and also to $F(\overline{e})$ where $\overline{e} = [n] - e$.

H_n also contains $2^n - 1$ "false" vertices, f_e, one for each $e \in X_n$. Node f_e starts with the path $S(e)$, and then from the last (blank) vertex — call it b'_e — there is an edge to each first vertex of $F(f)$, for any $f \in X_n$ such that $e \cup f \neq [n]$.

Let $T_n = \{t_e \mid e \in X_n\}$; $F_n = \{f_e \mid e \in X_n\}$. Clearly $T_n \models \text{Occur}_n$ and $F_n \models \neg\text{Occur}_n$.

Lemma 7.2 *Samson cannot close the $\mathcal{RL}^w_k$ game on (T_n, F_n) in a game tree with fewer than $2^n/2^k$ nodes.*

Proof: Note that the paths from t_e and f_e are identical through the blank vertices b_e, b'_e at the bottom of their starting paths, $S(e)$, and the only difference after that is that b_e has an edge to $F(\overline{e})$. Thus, to close the game tree, Samson must play a series of Reach moves from t_e to b_e, and then into $F(\overline{e})$ for each $e \in X_n$.

The key observation is that while we are standing on b_e, all that we know is what node of the game tree we are in, plus the current values of our k booleans. Indeed, we prove that Samson cannot play a REACH move that includes a path in which $(b_e, \overline{e})$ is an intermediate node, and also includes a path in which $(b_g, \overline{e})$ is an intermediate node, for distinct subsets $e \neq g$ and the same k-tuple of booleans $\overline{e}$. It follows that Samson can move through at most 2^k different b_e's at the same time. Our lower bound will then follow.

Suppose for the sake of a contradiction that for distinct subsets $e, g \in X_n$, Samson plays a Reach move that includes a step from from b_e and from b_g at the same node of the game tree and that the booleans associated with b_e and b_g are identical.

Since $e \neq g$ we may assume that $e \cup \overline{g} \neq [n]$, otherwise interchange e and g. Delilah answers with a Reach path from f_e to b'_e that first copies the booleans on Samson's path from t_e to b_e. Delilah continues this path to $F(\overline{g})$ copying Samson's path from b_g to $F(\overline{g})$. Since each step in Delilah's spliced path is identical to a step in one of Samson's paths, Samson cannot challenge any of the steps. Thus, Samson must challenge the bottom of Delilah's path. However this is identical to the bottom of Samson's path from t_g.

Thus our assumption was false, so at most 2^k t_e's can move from their blank vertices, b_e, at the same node of the game tree. Thus there must be at least $(2^n - 1)/2^k$ intermediate nodes of the game tree. Since there are at least n leaves, the total number of nodes is at least $2^n/2^k$ as claimed. □

Corollary 7.3 *$\Omega(n)$ booleans are required to express the* CTL$^+$ *and* LTL *formula* Occur_n *as a polynomial-size formula of $\mathcal{RL}^w$.*

8 Conclusions and Future Directions

In this paper we have introduced Ehrenfeucht-Fraïssé games on the size of formulas rather than their operator depth. We have used these games to prove a new, optimal bound which exactly characterizes the succinctness of CTL$^+$ with respect to CTL. We have also used these games to prove an $\Omega(n)$ lower bound on the number of booleans needed to translate LTL to $\mathcal{RL}^w$.

The formula-size games introduced here offer promise in settling many conjectures in descriptive complexity. In particular, questions about true complexity involve languages where an ordering relation on the universe is present. In the presence of ordering, we can express complex properties using low operator depth, with huge disjunctions over all possible input structures of a given size. Thus bounds on operator depth are not helpful here. Bounds on size would be extremely helpful. The formulas involved must be large, assuming well-believed complexity-theoretic conjectures. Although the size game is combinatorially complex, we expect that the methods introduced in this paper will help make progress towards lower bounds for languages with ordering.

We expect that the lower bounds from Section 7 can be extended to the full reachability logic, $\mathcal{RL}$. Another open problem was suggested by one of the referees: Wilke showed his exponential lower bound for the alternation-free μ-calculus which properly contains CTL [Wil99]. Can our Theorem 6.1 be similarly extended to the alternation-free μ-calculus?

Acknowledgments: Thanks to Natasha Alechina and Thomas Wilke for many helpful comments and suggestions.

References

[AI00] N. Alechina and N. Immerman, "Reachability Logic: An Efficient Fragment of Transitive Closure Logic," *Logic Journal of the IGPL* 8(3) (2000), 325-338.

[CE81] E.M. Clarke and E.A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," in *Proc. Workshop on Logic of Programs*, LNCS 131, 1981, Springer-Verlag, 52–71.

[CGP99] E. Clarke, O. Grumberg and D. Peled, *Model Checking*, 1999, M.I.T. press,

[Ehr61] A. Ehrenfeucht, "An Application of Games to the Completeness Problem for Formalized Theories," Fund. Math. 49 (1961), 129-141.

[EH85] E.A. Emerson and J.Y. Halpern, "Decision Procedures and Expressiveness in the Temporal Logic of Branching Time," *J. Comput. Sys. Sci.* 30(1) (1985), 1–24.

[EI95] K. Etessami and N. Immerman, "Tree Canonization and Transitive Closure," to appear in *Information and Computation* . A preliminary version appeared in *IEEE Symp. Logic In Comput. Sci.* (1995), 331-341.

[EW96] K. Etessami and T. Wilke, "An Until Hierarchy for Temporal Logic," *IEEE Symp. Logic In Comput. Sci.* (1996).

[Fra54] R. Fraïssé, "Sur les Classifications des Systems de Relations," Publ. Sci. Univ. Alger I (1954).

[Imm99] N. Immerman, *Descriptive Complexity*, 1999, Springer Graduate Texts in Computer Science, New York.

[Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[IV97] N. Immerman and M.Y. Vardi. Model Checking and Transitive Closure Logic. *Proc. 9th Int'l Conf. on Computer-Aided Verification* (CAV'97), Lecture Notes in Computer Science, Springer-Verlag 291 - 302, 1997.

[Kar89] M. Karchmer, *Communication Complexity: A New Approach to Circuit Depth,* 1989, M.I.T. Press.

[SC85] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *JACM,* 32(3), 733-749, 1985.

[Wil99] T. Wilke, "CTL$^+$ is Exponentially More Succinct than CTL", *Foundations of Software Technology and Theoretical Computer Science: 19th Conference*, (1999), 110–121.

A Background on CTL

A popular and quite expressive language for Model Checking is computation tree logic CTL* [CGP99]. Here we briefly describe CTL* together with some of its sublanguages: CTL $\subseteq$ CTL$^+$ $\subset$ CTL* and LTL $\subset$ CTL*. CTL and CTL$^+$ express the same set of formulas, but CTL$^+$ is more succinct. CTL and LTL are incomparable.

CTL* has two kinds of formulas: *state formulas*, which are true or false at each state, and *path formulas*, which are true or false with respect to a maximal path through some Kripke structure, $\mathcal{K}$. The following is an inductive definition of the state and path formulas of CTL*.

Definition A.1 (Syntax of CTL***)** State formulas $\mathcal{S}$ and path formulas $\mathcal{P}$ of CTL* are the smallest sets of formulas satisfying the following:
State Formulas, $\mathcal{S}$:
the boolean constants **true** and **false** are elements of $\mathcal{S}$;
for $i \in \mathbf{N}$, $p_i \in \mathcal{S}$;
if $\varphi \in \mathcal{P}$, then $\mathbf{E}\varphi \in \mathcal{S}$.

Intuitively, $\mathbf{E}\varphi$ means that there exists a maximal path starting at the current state and satisfying φ.
Path Formulas, $\mathcal{P}$:
if $\alpha \in \mathcal{S}$ then $\alpha \in \mathcal{P}$;
if $\varphi, \psi \in \mathcal{P}$, then $\neg\varphi$, $\varphi \wedge \psi$, $\mathbf{X}\varphi$, and $\varphi\mathbf{U}\psi$ are in $\mathcal{P}$.

Intuitively, $\mathbf{X}\varphi$ means that φ holds at the next time and $\varphi\mathbf{U}\psi$ means that at some time now or in the future, ψ holds, and from now until then, φ holds. □

Next, we formally define the semantics of the above operators. In this paper all structures will be finite and acyclic except perhaps for self-loops. Thus all paths will be finite, except perhaps for an infinite loop on the final point. A maximal path $\rho = \rho_1, \rho_2, \ldots \rho_\ell$ is a mapping from $[\ell]$ to states in $\mathcal{K}$ such that for all $i < \ell$, $\mathcal{K} \models R(\rho_i, \rho_{i+1})$ and such that ρ_ℓ either has no successors or it has a self-loop. We use the notation ρ^i for the tail of ρ, with states $\rho_1, \rho_2, \ldots, \rho_{i-1}$ removed.

Definition A.2 (Semantics of CTL***)** The following are inductive definitions of the meaning of CTL* formulas:
State Formulas:

$$(\mathcal{K}, s) \models p_i \quad \text{iff} \quad \mathcal{K} \models p_i(s)$$
$$(\mathcal{K}, s) \models \mathbf{E}\varphi \quad \text{iff} \quad (\exists \text{ path } \rho \text{ s.t. } \rho_0 = s)(\mathcal{K}, \rho) \models \varphi$$

Path Formulas:

$$(\mathcal{K}, \rho) \models \alpha \quad \text{iff} \quad (\mathcal{K}, \rho_0) \models \alpha; \text{ for } \alpha \in \mathcal{S}$$
$$(\mathcal{K}, \rho) \models \varphi \wedge \psi \quad \text{iff} \quad (\mathcal{K}, \rho) \models \varphi \text{ and } (\mathcal{K}, \rho) \models \psi$$
$$(\mathcal{K}, \rho) \models \neg\varphi \quad \text{iff} \quad (\mathcal{K}, \rho) \not\models \varphi$$
$$(\mathcal{K}, \rho) \models \mathbf{X}\varphi \quad \text{iff} \quad (\mathcal{K}, \rho^1) \models \varphi$$
$$(\mathcal{K}, \rho) \models \varphi\mathbf{U}\psi \quad \text{iff} \quad (\exists i)(\mathcal{K}, \rho^i) \models \psi \wedge (\forall j < i)(\mathcal{K}, \rho^j) \models \varphi$$

□

It is convenient to introduce a few other operators commonly used in CTL* all of which may be defined from the above:

$\mathbf{A}\varphi$	$\equiv$	$\neg\mathbf{E}\neg\varphi$	for All paths
$\mathbf{F}\varphi$	$\equiv$	$\mathbf{true}\mathbf{U}\varphi$	some time in the Future
$\mathbf{G}\varphi$	$\equiv$	$\neg\mathbf{F}\neg\varphi$	Globally, i.e., for all times in the future

The language CTL is the restriction of CTL* so that path quantifiers ($\mathbf{E}, \mathbf{A}$) and temporal operators ($\mathbf{X}, \mathbf{U}$) are always

paired. That is, the allowable operators are **EU**, **AU**, **EX**[4]. The importance of CTL is that unlike CTL* it admits linear-time model checking [CE81]. The language CTL$^+$ allows boolean combinations of the temporal operators to be paired with the path quantifiers. CTL$^+$ is no more expressive than CTL but it is more succinct [Wil99]. Our main result shows exactly how succinct. The language LTL (linear temporal logic) consists of CTL* formulas that have exactly one path quantifier **E** or **A** and that begin with this path quantifier.

B Background on $\mathcal{RL}$

Here we give the definition of Reachability Logic ($\mathcal{RL}$). See [AI00] for proofs of the theorems and much more motivation and discussion.

Definition B.1 An *adjacency formula* (with booleans) is a disjunction of conjunctions where each conjunct contains at least one of $x = y$, $R_a(x, y)$ or $R_a(y, x)$ for some edge label a; in addition, the conjuncts may contain expressions of the form $(\neg)(b_1 = b_2)$, $(b_1 = 0)$, $(b_1 = 1)$ and $p(x)$, where b_1 and b_2 are boolean variables. □

Definition B.2 $\mathcal{RL}$ is the smallest fragment of FO^2(TC) that satisfies the following:

1. If p is a unary relation symbol then $p \in \mathcal{RL}$; also $\top, \bot \in \mathcal{RL}$.
2. If $\varphi, \psi \in \mathcal{RL}$, then $\neg\varphi \in \mathcal{RL}$ and $\varphi \wedge \psi \in \mathcal{RL}$.
3. If $\varphi \in \mathcal{RL}$ and b is a boolean variable, then $\exists b\varphi \in \mathcal{RL}$.
4. If $\varphi, \psi \in \mathcal{RL}$ and q is a new unary predicate symbol, then (**let** $q = \varphi$ **in** ψ) is in $\mathcal{RL}$.
5. If $\varphi \in \mathcal{RL}$ and $\delta(x, \bar{b}, y, \bar{b'})$ is an adjacency formula (a binary relation between two n-tuples $\langle x, b_1, \ldots, b_{n-1}\rangle$ and $\langle y, b'_1, \ldots, b'_{n-1}\rangle$), then the following formulas are in $\mathcal{RL}$:
   (a) REACH$(\delta)\varphi$
   (b) CYCLE(δ)

Semantics of $\mathcal{RL}$: The semantics of $\mathcal{RL}$ is defined as follows. In each case below assume that $\delta(x, \bar{b}, y, \bar{b'})$ is an adjacency formula.

$$
\begin{aligned}
p &\equiv p(x) \\
(\textbf{let } q = \varphi \textbf{ in } \psi) &\equiv \psi[\varphi/q] \\
\text{REACH}(\delta)\varphi &\equiv \exists y(\text{TC}\,\delta)(x, \bar{0}, y, \bar{1}) \wedge \varphi[y/x]) \\
\text{CYCLE}(\delta) &\equiv (\text{TC}^s\,\delta)(x, \bar{0}, x, \bar{1})
\end{aligned}
$$

[4]We do not need **AX** because it is equivalent to $\neg$**EX**$\neg$

□

Here are some examples of formulas in $\mathcal{RL}$:

- REACH$(\delta)p$ where $\delta(x, b_1, b_2, y, b'_1, b'_2)$ is $(R_a(x, y) \wedge b_1b_2 = 00 \wedge b'_1b'_2 = 01) \vee (R_b(x, y) \wedge b_1b_2 = 01 \wedge b'_1b'_2 = 11)$ (this is $\langle a; b\rangle p$ of PDL).
- φ_1 = REACH$(R)p$ (**EF**p of CTL*);
- φ_2 = REACH(δ)CYCLE(δ), where δ is $R(x, y) \wedge q(x)$ (**EG**q of CTL*);
- (**let** $q = \varphi_1$ *in* φ_2) (**EGEF**p of CTL*).

$\mathcal{RL}$ is a logical language and it is a fragment of FO^2(TC). However, because of the 'let' construct, when we talk about size in the representation of $\mathcal{RL}$, we are really talking about circuits. Thus the size of an $\mathcal{RL}$-circuit may be logarithmic in the size of the smallest equivalent FO^2(TC) formula. This allows the linear size embedding of CTL* which presumably does not hold for FO^2(TC) (without a circuit representation or an extra domain variable cf. [IV97]).

Boolean variables however add extra complexity, which is not surprising since model checking CTL* is PSPACE complete [SC85].

Theorem B.3 *There is an algorithm that given a graph G and a formula $\varphi(x) \in \mathcal{RL}$ marks the vertices in G that satisfy φ. This algorithm runs in time $O(|G||\varphi|2^{n_b})$ where n_b is the number of boolean variables occurring in φ.*

Theorem B.4 *There is a linear-time computable function g that maps any* CTL* *formula φ to an equivalent formula $g(\varphi) \in \mathcal{RL}$. While $g(\varphi)$ has only two domain variables, it may have a linear number of boolean variables.*

Session 6

Light Affine Lambda Calculus and Polytime Strong Normalization

Kazushige Terui*
Department of Philosophy, Keio University
2-15-45 Mita, Minato-ku, Tokyo 108, Japan.
E-mail: terui@abelard.flet.keio.ac.jp

Abstract

Light Linear Logic (**LLL**) *and its variant,* Intuitionistic Light Affine Logic (**ILAL**)*, are logics of the polytime computation. It has been proved that all polynomial time functions are representable by proofs of these logics (via the proofs-as-programs correspondence), and conversely that there is a specific reduction (cut-elimination) strategy which normalizes a given proof in polynomial time (the latter may well be called the polytime "weak" normalization theorem).*

In this paper, we introduce an untyped term calculus, called Light Affine Lambda Calculus (λLA)*, generalizing the essential ideas of light logics into an untyped framework. It is a simple modification of λ-calculus, and has* **ILAL** *as a type assignment system. Then, in this generalized setting, we prove the polytime "strong" normalization theorem:* any *reduction strategy normalizes a given* λLA *term (of fixed depth) in a polynomial number of reduction steps, and indeed in polynomial time.*

1 Introduction

In [9, 10], Girard introduced *Light Linear Logic* (**LLL**) as an intrinsically polytime logical system: every polynomial time function is representable by an **LLL** proof, and every **LLL** proof[1] is normalizable, via cut-elimination, in polynomial time. Later on, in [2], Asperti introduced a simplified system, called *Light Affine Logic*, by adding the full (unrestricted) weakening rule to **LLL**. Its intuitionistic fragment (**ILAL**) has been particularly well investigated (see [3]), since it allows a compact term notation for proofs and has clear relevance to functional programming issues.

These light logics provide a *purely logical* insight into the polytime computation. In contrast with other polytime logical (type) systems, e.g., [15, 13, 11, 8, 14], light logics do not contain any built-in data type, and the characterization result is about the complexity of cut-elimination, which has been a canonical measure for estimating the complexity of a logical system in proof theory. Also notably, light logics are endowed with various semantics ([12, 4, 18]), which could lead to a semantic understanding of polytime.

An important problem remains to be settled, however. By inspecting the normalization theorem given by [10], one observes that what is actually shown in that paper is the polytime *weak* normalizability, namely, that there is a *specific* reduction strategy which normalizes a given **LLL** proof in polytime. The same is true of **ILAL** ([2, 20, 3]). It has been left unsettled whether the polytime *strong* normalizability holds for these light logics, namely, whether *any* reduction strategy normalizes a given proof in polytime. The primary purpose of this paper is to give a solution to this problem.

Having such a property will be theoretically important in that it gives further credence to light logics as intrinsically polytime systems. It will be practically important, too. Through the Curry-Howard correspondence, each proof of light logics may be considered as a *feasible program*, which is executable in polytime, and whose bounding polynomial is specified by its type (formula). In this context, the property will assure that the polytime executability of such a program is not affected by the choice of an evaluation strategy. It will also provide a good starting point for pursuit of efficiency in normalization.

For our purpose, it is reasonable to begin with **ILAL**, because it is much simpler than **LLL**. However, the term calculi proposed for **ILAL** so far either have a complicated notion of reduction defined by a large number of rewriting rules ([2, 20]), or involve notational ambiguity ([19, 3]).[2] Therefore, we first need to devise a simple and accurate term calculus for

*Research Fellow of the Japan Society for the Promotion of Science.

[1] Of lazy conclusions, i.e., those free from ∃ and &.

[2] See the remark in 9.1 of [3]. Instead, the latter paper presents a proofnet syntax for **ILAL**, based on which several computational properties are investigated.

0-7695-1281-X/01 $10.00 © 2001 IEEE

ILAL which is suitable for our investigation. Such a simple calculus will also provide a better understanding of the computational aspects of light logics. This is our secondary purpose.

In this paper, we introduce a new term calculus, called *Light Affine Lambda Calculus* (λLA), which embodies the essential mechanisms of light logics in an untyped setting. It amounts to a simple modification of λ-calculus with modal and let operators, having very simple operational behavior defined by just 5 reduction rules with the standard notion of substitution. It satisfies the subject reduction and Church-Rosser properties. λLA is an untyped calculus, but remarkably, all its *well-formed* terms are polytime normalizable. **ILAL** is then re-introduced as a Curry-style type assignment system for λLA. There are a number of reasons for adopting this presentation.

1. First of all, to design a truly polytime (rather than just polystep) polymorphic calculus, one must give up a Church-style term syntax with embedded types: a universal quantifier may bind an arbitrary number of type variable occurrences, and thus iterated type instantiations (Λ reductions) may easily cause exponential growth in the size of types.[3]
2. An untyped polytime calculus deserves investigation in its own right. (This program was advocated in the appendix of [10], but has not been developed so far.)
3. The notion of well-formedness, rather than typability, neatly captures the syntactic conditions for being polytime normalizable.
4. Last but not least, typability in **ILAL** is presumably intractable,[4] while well-formedness is checked very easily (in quadratic time).

Then, in this generalized setting, we prove

- The Polystep Strong Normalization Theorem: every reduction sequence in λLA has a length bounded by a polynomial in the size of its initial term (of fixed depth).
- The Polytime Strong Normalization Theorem: every reduction strategy (given as a function oracle) induces a normalization procedure which terminates in time polynomial in the size of a given term (of fixed depth).

It follows that every term typable in **ILAL**, which can be viewed as a structural representation of an **ILAL** proof (with formulas erased), is polytime strongly normalizable. It is very likely that essentially the same holds for **LLL**.

The rest of this paper is organized as follows. We introduce λLA in Section 2 and **ILAL** (as a type assignment system) in Section 3. In Section 4 we give the main part of the *polystep* strong normalization theorem. The theorem itself appears in Section 5, as well as its direct corollaries, namely the Church-Rosser property and the *polytime* strong normalization theorem. In Section 6 we discuss the polytime strong normalizability of **LLL**. We also discuss the interpretability of polytime type systems based on safe recursion in λLA.

2 Light Affine Lambda Calculus

In this section we set up λLA. We begin by giving the set $\mathcal{PT}$ of pseudo-terms (in 2.1). Our goal is to define the set $\mathcal{T}$ of well-formed terms (in 2.2) and the notion of reduction (in 2.3).

2.1 Pseudo-terms

Let $x, y, z \ldots$ range over term variables.

Definition 2.1 The set $\mathcal{PT}$ of *pseudo-terms* is defined by the following grammar:

$$t, u \quad ::= \quad x \mid \lambda x.t \mid tu \mid !t \mid \mathsf{let}\ u\ \mathsf{be}\ !x\ \mathsf{in}\ t \mid \S t \mid \mathsf{let}\ u\ \mathsf{be}\ \S x\ \mathsf{in}\ t.$$

In addition to the standard constructs such as λ-abstraction and application, we have two *boxes*, $!t$ and $\S t$, and two let *operators*. Boxes induce a stratified structure on expressions. Interaction of boxes is enabled by let operators.

In the sequel, symbol $\dagger$ stands for either ! or §. Pseudo-terms $(\lambda x.t)$ and $(\mathsf{let}\ u\ \mathsf{be}\ \dagger x\ \mathsf{in}\ t)$ *bind* each occurrence of x in t. As usual, pseudo-terms are considered up to α-equivalence, and the *variable convention* (see [5]) is adopted for the treatment of free/bound variables (namely, the bound variables are chosen to be different from the free variables, so that variable clash is never caused by substitution). Notation $t\{u/x\}$ is used to denote the pseudo-term obtained by substituting u for the free occurrences of

[3] Proofnets (of **LLL**) contain formulas. Hence proofnets themselves are not polytime normalizable. A solution suggested by [10] is to work with untyped proofnets (with formulas erased) in the actual computation. When the conclusion is lazy, the formulas can be automatically recovered after normalization, and such formulas are not exponentially large. Our approach is somewhat similar, in that we work with an untyped formalism in the actual computation and supply it with a type assignment system.

[4] The problem is undecidable for System F in the Curry style ([22]).

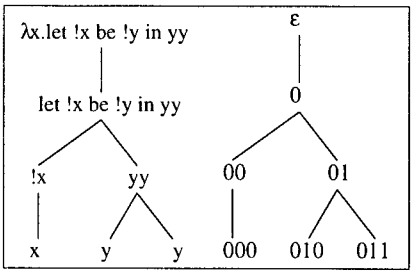


Figure 1: Term Tree and Addresses

x in t. $FV(t)$ denotes the set of free variables in t. $FO(x,t)$ denotes the number of free occurrences of x in t and $FO(t)$ denotes the number of free occurrences of all variables in t.

As usual, each pseudo-term t is represented as a *term tree*, and each subterm occurrence u in t is pointed by its *address*, i.e., a word $w \in \{0,1\}^*$ which describes the path from the root to the node corresponding to u in the term tree. For example, the term tree for $(\lambda x.\mathsf{let}\ !x\ \mathsf{be}\ !y\ \mathsf{in}\ yy)$ and the addresses in it are illustrated in Figure 1.

The *size* $|t|$ of a pseudo-term t is the number of nodes in its term tree. Since our terms are untyped, $|t|$ is not significantly different from the length of its string representation. Given a pseudo-term t and an address w, the *depth* of w in t is the number of !-boxes and §-boxes enclosing the subexpression at w. The *depth* of t is the maximum depth of all addresses in it.

A *context* Φ is a pseudo-term-like expression with one hole $\bullet$. If Φ is a context and t is a pseudo-term, then $\Phi[t]$ denotes the pseudo-term obtained by substituting t for $\bullet$ in Φ.

2.2 Terms

Before giving the formal definition of well-formed terms, we shall informally discuss the critical issues.

Firstly, we assume that variables are (conceptually) classified into three groups: undischarged, !-discharged, and §-discharged variables. These are to be bound by λ-abstraction, let-! operator and let-§ operator, respectively.

The fundamental concept of light logics is *to enforce a stratified structure on proofs/terms and to preserve it in the course of reduction.* Concretely, light logics deny the following principles of Linear Logic which destroy the stratified structure:

- *Dereliction*: $!A \multimap A$,
- *Digging*: $!A \multimap !!A$.

We achieve the stratification by the following mechanisms:

- *In default, a variable is undischarged, and a variable is made (either !- or §-) discharged when a box is built around it.* This condition corresponds to the prohibition of the dereliction principle. It is expressed in our term syntax as:

  $$\mathit{dereliction}(x) := \mathsf{let}\ x\ \mathsf{be}\ !y\ \mathsf{in}\ y,$$

  whose effect is to open a !-box:

  $$\mathit{dereliction}(!t) \longrightarrow t.$$

  It is ruled out, since variable y is undischarged, but is illegally bound by a let-! operator. On the other hand, the following term corresponding to the canonical map $!A \multimap \S A$ is legitimated:

  $$\mathsf{let}\ x\ \mathsf{be}\ !y\ \mathsf{in}\ \S y.$$

- *A box may be built around a term only when it contains no discharged variable.* This corresponds to the prohibition of the digging principle. It may be expressed as:

  $$\mathit{digging}(x) := \mathsf{let}\ x\ \mathsf{be}\ !y\ \mathsf{in}\ !!y,$$

  whose effect is to embed a !-box into a deeper layer:

  $$\mathit{digging}(!t) \longrightarrow !!t.$$

  It is also ruled out, since it attempts to build a !-box $!!y$ around another box $!y$, but the latter contains a discharged variable y.

Another fundamental property of light logics is, as in Linear Logic, that *only duplicable are contents of !-boxes.* It is maintained by the following condition:

- *Among three binders, only* let-*! may bind multiple occurrences of (!-discharged) variables.*

Duplication takes place when a !-box meets a let-! operator; for example,

$$\mathsf{let}\ !t\ \mathsf{be}\ !x\ \mathsf{in}\ (\S xx)!x \longrightarrow (\S tt)!t.$$

To avoid potential exponential growth caused by duplication, we need a further constraint on !-boxes:

- *A !-box may be built around a term only when it contains at most one free variable.*

Hence term constructions like

$$\ldots \text{let } !zz \text{ be } !y \text{ in } (\text{let } !yy \text{ be } !x \text{ in } !xx),$$

which cause exponential growth are prohibited.

To compensate for this, we need another kind of boxes, namely §-boxes. They are not duplicable. Instead they may contain an arbitrary number of free variables.

All these design concepts (and more) are realized in the following formal definition, which is written in a style inspired by [1].

Definition 2.2 Let X, Y, Z range over the finite sets of variables. Then the 4-ary relation $t \in \mathcal{T}_{X,Y,Z}$ (saying that t is a (well-formed) term with undischarged variables X, !-discharged variables Y and §-discharged variables Z) is defined as follows (in writing $t \in \mathcal{T}_{X,Y,Z}$, we implicitly assume that X, Y and Z are mutually disjoint):

1. $x \in \mathcal{T}_{X,Y,Z} \iff x \in X$.
2. $\lambda x.t \in \mathcal{T}_{X,Y,Z} \iff$
   $t \in \mathcal{T}_{X\cup\{x\},Y,Z},\ x \notin X,\ FO(x,t) \leq 1$.
3. $tu \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{X,Y,Z},\ u \in \mathcal{T}_{X,Y,Z}$.
4. $!t \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{Y,\emptyset,\emptyset},\ FO(t) \leq 1$.
5. $\S t \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{Y\cup Z,\emptyset,\emptyset}$.
6. $\text{let } t \text{ be } !x \text{ in } u \in \mathcal{T}_{X,Y,Z} \iff$
   $t \in \mathcal{T}_{X,Y,Z},\ u \in \mathcal{T}_{X,Y\cup\{x\},Z},\ x \notin Y$.
7. $\text{let } t \text{ be } \S x \text{ in } u \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{X,Y,Z}$,
   $u \in \mathcal{T}_{X,Y,Z\cup\{x\}},\ x \notin Z,\ FO(x,u) \leq 1$.

Finally, t is a *(well-formed) term* ($t \in \mathcal{T}$) if $t \in \mathcal{T}_{X,Y,Z}$ for some X, Y and Z.

Example 2.3

1. $\omega_{LA} \equiv \lambda x.(\text{let } x \text{ be } !y \text{ in } \S yy) \in \mathcal{T}$,
   $\Omega_{LA} \equiv \omega_{LA}!\omega_{LA} \in \mathcal{T}$.
2. For each natural number n, we have Church numeral $\overline{n} \in \mathcal{T}$ defined by
   $$\overline{n} \equiv \lambda x.(\text{let } x \text{ be } !z \text{ in } \S\lambda y.\underbrace{(z \cdots (z}_{n \text{ times}} y)\cdots)).$$
3. For each word $w \equiv i_0 \cdots i_n \in \{0,1\}^*$, we have $\overline{w} \in \mathcal{T}$ defined by
   $$\overline{w} \equiv \lambda x_0 x_1.(\text{let } x_0 \text{ be } !z_0 \text{ in } (\text{let } x_1 \text{ be } !z_1 \text{ in } \S\lambda y.(z_{i_0} \cdots (z_{i_n} y)\cdots))).$$
   Observe that these $\overline{n}$'s and $\overline{w}$'s are all of depth 1.

We have the following basic properties:

Lemma 2.4 *Let $t \in \mathcal{T}_{X,Y,Z}$.*

1. *If $X \subseteq X'$, $Y \subseteq Y'$ and $Z \subseteq Z'$, then $t \in \mathcal{T}_{X',Y',Z'}$.*
2. *If $x \notin FV(t)$, then $t \in \mathcal{T}_{X\setminus\{x\},Y\setminus\{x\},Z\setminus\{x\}}$.*
3. *Let $x \in FV(t)$. Then x occurs at depth 0 iff $x \in X$. x occurs at depth 1 iff $x \in Y \cup Z$. x never occurs at depth > 1.*

Lemma 2.5 (Substitution)

1. $t \in \mathcal{T}_{X\cup\{x\},Y,Z}$, $x \notin X$ *and* $u \in \mathcal{T}_{X,Y,Z}$
   $\Longrightarrow t\{u/x\} \in \mathcal{T}_{X,Y,Z}$.
2. $t \in \mathcal{T}_{X,Y\cup\{x\},Z}$, $x \notin Y$ $u \in \mathcal{T}_{Y,\emptyset,\emptyset}$ *and* $FO(u) \leq 1$
   $\Longrightarrow t\{u/x\} \in \mathcal{T}_{X,Y,Z}$.
3. $t \in \mathcal{T}_{X,Y,Z\cup\{x\}}$, $x \notin Z$ *and* $u \in \mathcal{T}_{Y\cup Z,\emptyset,\emptyset}$
   $\Longrightarrow t\{u/x\} \in \mathcal{T}_{X,Y,Z}$.

Remark 2.6 As discussed by Asperti [2], a naive use of box notation causes ambiguity, and in conjunction with naive substitutions, causes a disastrous effect on complexity.

Asperti fixed the by using a more sophisticated box notation of the form $\S(t)[u_1/x_1, \ldots, u_n/x_n]$, while our solution is more implicit and is based on the conceptual distinction between discharged and undischarged variables.

Asperti's box $\S(tx_1x_2)[y/x_1, y/x_2]$ (with y of !-type) corresponds to $(\text{let } y \text{ be } !x \text{ in } \S(txx))$ in our syntax. Observe that variable y, which is external to the §-box, is contracted in the former, while variable x, which is internal to the §-box, is contracted in the latter. This is parallel to the difference between the contraction inference rule of Asperti's **ILAL** and that of Girard's original formation of **LLL**; the former contracts !-formulas, while the latter contracts discharged formulas.

Remark 2.7 There is a quadratic time algorithm checking whether a given pseudo-term is well-formed: Let t be a pseudo-term, and X and Y be the sets of its free variables at depth 0 and at depth 1, respectively. Then t is well-formed iff $t \in \mathcal{T}_{X,Y,\emptyset}$ (by Lemma 2.4 and the fact that $t \in \mathcal{T}_{X,Y,Z}$ implies $t \in \mathcal{T}_{X,Y\cup Z,\emptyset}$). The latter can be recursively checked with at most $|t|$ recursive calls, and each call involves a variable occurrence check at most once (corresponding to Clauses 2, 4 and 7 of Definition 2.2). Thus the algorithm runs in time $O(n^2)$, given a term of size n.

Name	*Redex*	*Contractum*
(β)	$(\lambda x.t)u$	$t\{u/x\}$
(§)	let §u be §x in t	$t\{u/x\}$
(!)	let !u be !x in t	$t\{u/x\}$
(*com*)	(let u be † x in t)v	let u be † x in (tv)
	let (let u be † x in t) be † y in v	let u be † x in (let t be † y in v)

Figure 2: Reduction Rules

2.3 Reduction

Definition 2.8 The *reduction rules* of λLA are those listed in Figure 2. We say that t *reduces to* u *at address* w *by rule* (r), and write as $t \xrightarrow{w,(r)} u$, if $t \equiv \Phi[v_1]$, $u \equiv \Phi[v_2]$, the hole $\bullet$ is located at w in Φ, and v_1 is an (r)-redex whose contractum is v_2.

Note that the address w uniquely determines the rule (r) to be used. When either the address w or the rule (r), or both, are irrelevant, we use notations $t \xrightarrow{(r)} u$, $t \xrightarrow{w} u$ and $t \longrightarrow u$. The *depth* of a reduction is the depth of its redex.

A finite sequence σ of addresses $w_0, \ldots, w_{n-1}$ is said to be a *reduction sequence* from t_0 to t_n, written as $t_0 \xrightarrow{\sigma} {}^* t_n$, if there are pseudo-terms $t_0, \ldots, t_n$ such that

$$t_0 \xrightarrow{w_0} t_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{n-1}} t_n.$$

If every reduction in σ is the application of (r), then σ is called an *(r)-reduction sequence* and written as $t_0 \xrightarrow{\sigma,(r)} {}^* t_n$ (or simply as $t_0 \xrightarrow{(r)} {}^* t_n$). The length of σ is denoted by $|\sigma|$.

Remark 2.9 The stratified structure of a term is preserved by reduction. In particular, the depth of a term never increases, since in reduction rules (β), (§) and (!) a subterm u is substituted for a variable x occurring at the same depth.

Reduction rules (β) and (§) strictly decrease the size of a term, since they never involve duplication. (*com*) just reorganaizes the structure of a term without changing its size. The only reduction rule which causes duplication is (!). When applied at depth i, it possibly increases the sizes at depths $> i$, while it strictly decreases the size at depth i.

The terms are closed under reduction:

Proposition 2.10 *If* $t \in \mathcal{T}_{X,Y,Z}$ *and* $t \longrightarrow u$, *then* $u \in \mathcal{T}_{X,Y,Z}$.

Proof. For example, if t is a (!) redex let !u be !x in v, then $v \in \mathcal{T}_{X,Y\cup\{x\},Z}$, $u \in \mathcal{T}_{Y,\emptyset,\emptyset}$ and $FO(u) \leq 1$. Hence $v\{u/x\} \in \mathcal{T}_{X,Y,Z}$ by Lemma 2.5. For the general case, show that a term $u \in \mathcal{T}_{X,Y,Z}$ can be replaced with another $v \in \mathcal{T}_{X,Y,Z}$ in a context without losing well-formedness, whenever $FO(x,v) \leq FO(x,u)$ for each $x \in X \cup Z$. All reduction rules $u \longrightarrow v$ meet the latter condition. ■

Example 2.11 The term Ω_{LA} is a light affine analogue of $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$, which is not normalizable in λ-calculus. However,

$$\begin{aligned} \Omega_{LA} \equiv \omega_{LA}!\omega_{LA} &\xrightarrow{(\beta)} (\text{let } !\omega_{LA} \text{ be } !y \text{ in } \S yy) \\ &\xrightarrow{(!)} \S\omega_{LA}\omega_{LA} \\ &\xrightarrow{(\beta)} \S(\text{let } \omega_{LA} \text{ be } !y \text{ in } \S yy). \end{aligned}$$

The last term cannot be reduced anymore.

3 Type Assignment System

We introduce **ILAL** as a type assignment system for λLA. Our formulation is, however, different from Asperti's in that we use Girard's discharged formulas.

Let α, β range over the type variables.

Definition 3.1 The *types* (*formulas*) of **ILAL** are given by the following grammar:

$$A, B ::= \alpha \mid A \multimap B \mid \forall\alpha.A \mid !A \mid \S A.$$

An *!-discharged* type is an expression of the form $[A]_!$. An *§-discharged* type is an expression of the form $[A]_\S$.

In the sequel, $\dagger^n A$ abbreviates $\underbrace{\dagger \cdots \dagger}_{n \text{ times}} A$.

A *declaration* is an expression of the form $x:A$ or $x:[A]_\dagger$. A finite set of declarations is denoted by Γ, Δ, etc.

Definition 3.2 The *type inference rules* of **ILAL** are those given in Figure 3. We say that a pseudo-term t is *typable* in **ILAL** if $\Gamma \vdash t:A$ is derivable for some Γ and A by those inference rules.

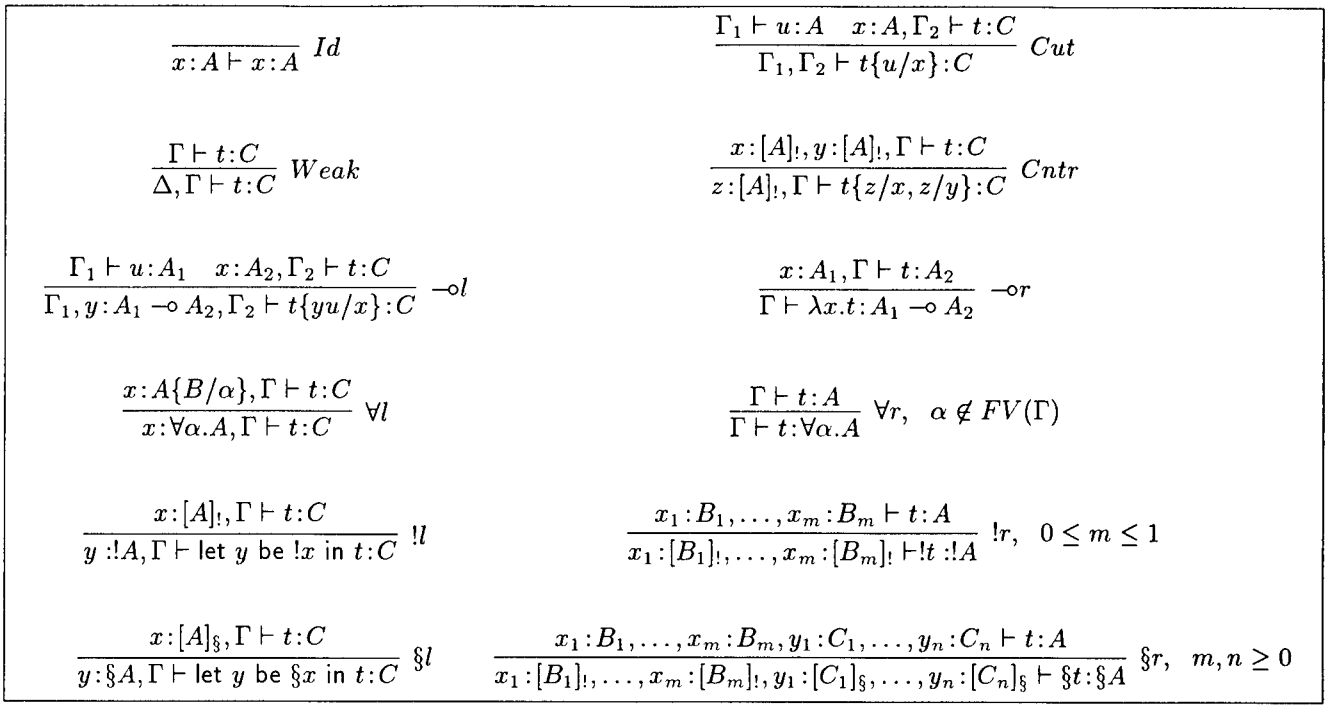


Figure 3: Type Assignment System **ILAL**

Remark 3.3 Observe that if $x:A,\Gamma \vdash t:C$, namely x is of undischarged type, then it occurs at most once in t. Therefore, no duplication is caused by the substitutions used in (Cut) and $(\multimap l)$ rules, which always operate on undischarged types. That is a reason why we can do away with explicit substitutions of [2].

Discharged types act as a *barrier* to substitution into boxes, in the same way as Wadler[21]'s *patterns* act in his term syntax for Intuitionistic Linear Logic; we could alternatively use the latter to obtain the same effect.

As expected, we have:

Theorem 3.4 *Every typable pseudo-term is a term. More exactly, if*

$$\vec{x}:\vec{A},\vec{y}:[\vec{B}]_!,\vec{z}:[\vec{C}]_\S \vdash t:D,$$

then $t \in \mathcal{T}_{\{\vec{x}\},\{\vec{y}\},\{\vec{z}\}}$.

Proof. By induction on the length of the typing derivation. In the cases of (Cut) and $(\multimap l)$, apply Lemma 2.5(1). ■

Theorem 3.5 (Subject Reduction) *If* $\Gamma \vdash t:A$ *and* $t \longrightarrow u$, *then* $\Gamma \vdash u:A$.

Example 3.6 Let $\mathbf{int} \equiv \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ and $\mathbf{bint} \equiv \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. Then we have $\vdash \overline{n}:\mathbf{int}$ for each $n \in N$ and $\vdash \overline{w}:\mathbf{bint}$ for each $w \in \{0,1\}^*$.

An example of untypable terms is Ω_{LA}. To see the reason, define the *erasure* of a term of λLA to be a λ-term obtained by applying the following operations as much as possible:

$$\begin{aligned} \dagger u &\mapsto u, \\ \mathsf{let}\ u\ \mathsf{be}\ \dagger x\ \mathsf{in}\ t &\mapsto t\{u/x\}. \end{aligned}$$

If a term is typable in **ILAL**, then its erasure is typable in System F (in the Curry style, see [6]). Now, Ω_{LA} cannot be typed in **ILAL**, since the erasure of Ω_{LA} is Ω, a term which cannot be typed in System F.

Remark 3.7 Types are not necessary for the polytime normalizability. Nevertheless, they are useful in several ways.

- Types are used to avoid *deadlocks*, such as $(\dagger t)u$ and $\mathsf{let}\ (\lambda x.t)\ \mathsf{be}\ \dagger x\ \mathsf{in}\ u$.
- Some types, typically data types such as **int** and **bint**, constrain the shape of normal forms: every normal term of type **int** is of the form $\overline{n}$ (or $\lambda x.(\mathsf{let}\ x\ \mathsf{be}\ !z\ \mathsf{in}\ \S z)$, which may be seen as an η-variant of $\overline{1}$). In general, for $k \ge 0$, every normal

term of type $\S^k\mathbf{int}$ is of the form $\S^k\overline{n}$ (or an η-variant of $\S^k\overline{1}$). Similarly, all normal inhabitants of **bint** are of the form $\overline{w}$.

- More generally, all lazy types, including **int** and **bint**, constrain the depths of normal forms: Say that a type is *lazy* if it does not contain a negative occurrence of $\forall$. If a term t is normal and of lazy type A, then it means that $\vdash t : A$ can be derived without using the $(\forall l)$ inference rule, which has an effect of hiding some information on derivations. Thus all uses of the ! and § inference rules in the derivation are recorded in A. Hence the depth of t is immediately bounded by the depth d of A.

- The above suggests that in order to normalize a term of lazy type A we do not have to fire redices at depth $> d$, which will be removed by reductions at lower depths before arriving at the normal form. In this way, lazy types give us useful information on normalization.

The expressive power of **ILAL**, hence of λLA, is witnessed by:

Theorem 3.8 (Girard[10], Roversi[19]) *Every function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ which is computable in time $O(n^d)$ is represented by a term of type* $\mathbf{bint} \multimap \S^{d+6}\mathbf{bint}$.

(See [3] for a good exposition. See also [17] for another proof).

The converse will be taken up in Section 5 after the polytime normalizability of λLA has been proved.

Remark 3.9 We are rather free in the choice of type systems; for example we can enrich **ILAL** with naive set theory or fixpoints of types (as in [10]), still preserving the polytime normalizability and the logical consistency (i.e., having no inhabitant of $\mathbf{0} \equiv \forall\alpha.\alpha$). To put it the other way round, *any* logical system which is cut-free consistent (i.e., with no normal inhabitant of **0**) is consistent, in so far as it can be used as a type system for λLA and satisfies the properties of Theorems 3.4 and 3.5.

4 Proving the Polystep Strong Normalization Theorem

The key step toward the polystep strong normalization theorem is the *standardization*, i.e., to transform a reduction sequence into an outer-layer-first one without decreasing the length (in 4.2). To achieve this, we first need to extend λLA with *explicit* weakening and to give a translation of reduction sequences in λLA into this extended calculus (in 4.1). Finally we show that the length of a standard reduction sequence thus obtained is polynomially bounded (in 4.3).

4.1 An extended calculus with explicit weakening

The set $\mathcal{PT}^w$ of *extended pseudo-terms* is defined analogously to $\mathcal{PT}$, but each extended pseudo-term may contain a subexpression of the form $\mathsf{let}\ t\ \mathsf{be}\ _\ \mathsf{in}\ u$ (explicit weakening). To define the well-formedness, we give a new 4-ary relation $t \in \mathcal{T}^w_{X,Y,Z}$ by modifying Definition 2.2 as follows.

(1) Replace clause 2, 6, and 7 with:

2' $\lambda x.t \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{X\cup\{x\},Y,Z}$, $x \notin X$, $FO(x,t) = 1$.

6' $\mathsf{let}\ t\ \mathsf{be}\ !x\ \mathsf{in}\ u \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{X,Y,Z}$, $u \in \mathcal{T}_{X,Y\cup\{x\},Z}$, $x \notin Y, FO(x,u) \geq 1$.

7' $\mathsf{let}\ t\ \mathsf{be}\ \S x\ \mathsf{in}\ u \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{X,Y,Z}$, $u \in \mathcal{T}_{X,Y,Z\cup\{x\}}$, $x \notin Z$, $FO(x,u) = 1$.

(Namely, we require that each binder must bind at least one variable occurrence.)

(2) Add the following clause:

8' $\mathsf{let}\ t\ \mathsf{be}\ _\ \mathsf{in}\ u \in \mathcal{T}_{X,Y,Z} \iff t \in \mathcal{T}_{X,Y,Z}$, $u \in \mathcal{T}_{X,Y,Z}$.

We say that t is a *(well-formed) extended term* ($t \in \mathcal{T}^w$) if $t \in \mathcal{T}^w_{X,Y,Z}$ for some X, Y, Z.

The reduction rules in Figure 2 are extended to $\mathcal{PT}^w$ with the following modifications:

- Generalize (*com*) so that it is also applicable to the new let operator for explicit weakening.

- Add a new reduction rule (_): $\mathsf{let}\ u\ \mathsf{be}\ _\ \mathsf{in}\ t \longrightarrow t$.

Reduction rules other than (_) are called *proper*. A reduction sequence is *proper* if every reduction in it is proper.

Lemmas 2.4 and 2.5 hold for $\mathcal{T}^w$, too. In addition, we have:

Proposition 4.1 *If $t \in \mathcal{T}^w_{X,Y,Z}$, $t \xrightarrow{(r)} u$ and (r) is proper, then $u \in \mathcal{T}^w_{X,Y,Z}$.*

Now we consider a translation of λLA into the extended calculus.

Lemma 4.2 *For each term t, there is an extended term t^w such that $t^w \xrightarrow{(_)}{}^* t$ and $|t^w| \leq 4|t|$.*

Proof. By induction on t. If $t \equiv \lambda x.u$ and $FO(x,u) = 0$, let $t^w \equiv \lambda x.(\mathsf{let}\ x\ \mathsf{be}\ _\ \mathsf{in}\ u^w)$. If $t \equiv (\mathsf{let}\ v\ \mathsf{be}\ \dagger x\ \mathsf{in}\ u)$ and $FO(x,u) = 0$, let $t^w \equiv \mathsf{let}\ v\ \mathsf{be}\ \dagger x\ \mathsf{in}\ (\mathsf{let}\ \S x\ \mathsf{be}\ _\ \mathsf{in}\ u^w)$. ■

Theorem 4.3 (Translation into the extended calculus) *Let t_0 be a term and let*

$$t_0 \xrightarrow{\sigma}{}^* t_1$$

be a reduction sequence in λLA. Then there are extended terms t_0', t_1' and a proper reduction sequence τ such that $|\sigma| \leq |\tau|$, $|t_0'| \leq 4|t_0|$ and

$$\begin{array}{ccc} t_0 & \xrightarrow{\sigma}{}^* & t_1 \\ {\scriptstyle (_)}^*\uparrow & & \uparrow{\scriptstyle (_)}^* \\ t_0' & \xrightarrow{\tau}{}^* & t_1'. \end{array}$$

Proof (Idea). By Lemma 4.2, there is an extended term t_0^w such that

$$t_0^w \xrightarrow{(_)}{}^* t_0 \xrightarrow{\sigma}{}^* t_1.$$

By permuting it suitably, we can obtain

$$t_0^w \xrightarrow{\tau}{}^* t_1' \xrightarrow{(_)}{}^* t_1,$$

such that τ is proper and $|\tau| \geq |\sigma|$. For example, a reduction sequence of the form

$$(\mathsf{let}\ v\ \mathsf{be}\ _\ \mathsf{in}\ (\lambda x.t))u \xrightarrow{(_)} (\lambda x.t)u \xrightarrow{(\beta)} t\{u/x\}$$

can be transformed into the following longer one:

$$(\mathsf{let}\ v\ \mathsf{be}\ _\ \mathsf{in}\ (\lambda x.t))u \xrightarrow{(com)} \mathsf{let}\ v\ \mathsf{be}\ _\ \mathsf{in}\ ((\lambda x.t)u)$$

$$\xrightarrow{(\beta)} \mathsf{let}\ v\ \mathsf{be}\ _\ \mathsf{in}\ t\{u/x\} \xrightarrow{(_)} t\{u/x\}.$$

In more detail, we use the following two lemmas for each step of permutation, which are shown by exhaustive case analyses. ■

Lemma 4.4 *Let $t_0 \in \mathcal{PT}^w$. If $t_0 \xrightarrow{(_)} t_1 \xrightarrow{(com)} t_2$, then*

$$t_0 \xrightarrow{\sigma,(com)_*} t_1' \xrightarrow{(_)} t_2$$

for some t_1' and $|\sigma| \geq 1$.

Lemma 4.5 *Let $t_0 \in \mathcal{PT}^w$. If $t_0 \xrightarrow{(_)} t_1 \xrightarrow{(r)} t_2$, where (r) is neither (com) nor $(_)$, then*

$$t_0 \xrightarrow{(com)_*} t_1' \xrightarrow{(r)} t_1'' \xrightarrow{(_)}{}^* t_2$$

for some t_1' and t_1''.

4.2 Standardization theorem

A reduction sequence σ is *standard* if it can be partitioned into subsequences $\sigma_0; \sigma_1; \ldots; \sigma_{2d}$, such that, for $i \leq d$, σ_{2i+1} consists of (!)-reductions at depth i and σ_{2i} consists of other reductions at depth i.

Theorem 4.6 (Standardization) *Let t_0 be an extended term and σ be a proper reduction sequence*

$$t_0 \xrightarrow{\sigma}{}^* t_1.$$

Then there is a standard proper reduction sequence τ

$$t_0 \xrightarrow{\tau}{}^* t_1$$

such that $|\sigma| \leq |\tau|$.

Proof (Idea). The proof is again based on permutaion of reduction sequences. For example, let u be a (β) redex and u' be its contractum, and consider the following nonstandard reduction sequence:

$$\mathsf{let}\ !u\ \mathsf{be}\ !x\ \mathsf{in}\ v \xrightarrow{(\beta)} \mathsf{let}\ !u'\ \mathsf{be}\ !x\ \mathsf{in}\ v \xrightarrow{(!)} v\{u'/x\}.$$

Here the first reduction is at depth 1 and the second at depth 0. It can be standardized as follows:

$$\mathsf{let}\ !u\ \mathsf{be}\ !x\ \mathsf{in}\ v \xrightarrow{(!)} v\{u/x\} \xrightarrow{\sigma,(\beta)_*} v\{u'/x\}.$$

Since $(\mathsf{let}\ !u\ \mathsf{be}\ !x\ \mathsf{in}\ v)$ is an extended term, we have $FO(x,v) \geq 1$. Hence $v\{u/x\}$ contains at least one occurrence of the (β) redex u, so $|\sigma| \geq 1$. Therefore the length of a reduction sequence never decreases by this permutation. ■

4.3 Bounding lengths of standard reduction sequences

For each extended term t its *partial size* $s_i(t)$ *at depth* i is defined in Figure 4 (where i ranges over the numbers ≥ 1).

We define $s(t)$ to be $\sum_{i=0}^{\infty} s_i(t)$. The only difference between $|t|$ and $s(t)$ is that the size of a box $\dagger t$ in the latter sense also counts the number of free variable occurrences in t. Note that $|t| \leq s(t) \leq 2|t|$.

The theorem below is essentially due to [10, 2]. In our case, however, the length of a reduction sequence may slightly exceed the size of its final term, since we have the commuting reduction rule (com).

$s_0(x)$	$=$	1		$s_i(x)$	$=$	0
$s_0(\lambda x.t)$	$=$	$s_0(t)+1$		$s_i(\lambda x.t)$	$=$	$s_i(t)$
$s_0(tu)$	$=$	$s_0(t)+s_0(u)+1$		$s_i(tu)$	$=$	$s_i(t)+s_i(u)$
$s_0(\dagger t)$	$=$	$FO(t)+1$		$s_i(\dagger t)$	$=$	$s_{i-1}(t)$
$s_0(\mathsf{let}\ t\ \mathsf{be}\ \dagger x\ \mathsf{in}\ u)$	$=$	$s_0(t)+s_0(u)+1$		$s_i(\mathsf{let}\ t\ \mathsf{be}\ \dagger x\ \mathsf{in}\ u)$	$=$	$s_i(t)+s_i(u)$
$s_0(\mathsf{let}\ t\ \mathsf{be}\ _\ \mathsf{in}\ u)$	$=$	$s_0(t)+s_0(u)+1$		$s_i(\mathsf{let}\ t\ \mathsf{be}\ _\ \mathsf{in}\ u)$	$=$	$s_i(t)+s_i(u)$

Figure 4: Partial Sizes

Theorem 4.7 (Polynomial bounds for standard reduction sequences) *Let t_0 be an extended term of depth d and σ be a standard proper reduction sequence $t_0 \xrightarrow{\sigma} {}^* u$. Then $s(u) \leq s(t_0)^{2^d}$ and $|\sigma| \leq s(t_0)^{2^{d+1}}$.*

Proof. The first claim is proved by iteratively applying Lemma 4.8 below, starting from depth 0 and ending with depth d. See also Remark 2.9. The second claim follows by Lemma 4.9. ■

Lemma 4.8 *Let σ be a reduction sequence $t \xrightarrow{\sigma} {}^* t'$ which consists of (!) reductions at depth i. Then we have $s_j(t') \leq s_j(t) \cdot s_i(t)$ for each $j > i$.*

Proof (Idea). For simplicity, let us assume $i = 0$ and $j = 1$. To estimate the potential size growth caused by (!) reductions, we make the following definition. For each extended term t, its *unfolding* is an extended pseudo-term $\sharp t \in \mathcal{PT}^w$ which is obtained by hereditarily replacing each subterm of the form $(\mathsf{let}\ !t\ \mathsf{be}\ !x\ \mathsf{in}\ u)$ at depth 0 with

$$\mathsf{let}\ \underbrace{!t!t\cdots!t}_{n\ times}\ \mathsf{be}\ !x\ \mathsf{in}\ \sharp u,$$

where $n = FO(x, \sharp u)$. (Intuitively, we perform all possible "contraction reductions" in advance.)

Then we can show

(1) $FO(\sharp v) \leq s_0(v)$,

(2) $s_1(v) \leq s_1(\sharp v) \leq s_0(v) \cdot s_1(v)$,

by induction on v. (The property that each !-box contains at most one free variable is crucial here.) Moreover, we can also show that

(3) if $v \xrightarrow{(!)} v'$ at depth 0, then $s_1(\sharp v') \leq s_1(\sharp v)$.

The lemma follows from (2) and (3):

$$s_1(t') \leq s_1(\sharp t') \leq s_1(\sharp t) \leq s_0(t) \cdot s_1(t).$$

■

Lemma 4.9 *Let σ be a reduction sequence $t \xrightarrow{\sigma} {}^* t'$ which consists of reductions at depth i. Then we have $|\sigma| \leq s_i(t)^2$.*

Proof (Idea). For simplicity, assume that $i = 0$. Let v be an extended term. For each occurrence of a let subterm $u \equiv (\mathsf{let}\ u_1\ \mathsf{be}\ *\ \mathsf{in}\ u_2)$ at depth 0 in v, where $*$ is either $_$ or $\dagger x$, define

$$com(u, v) := s_0(v) - s_0(u_2).$$

Define $com(v)$ to be the sum of all $com(u, v)$'s with u ranging over all such occurrences of let-expressions. Then we claim:

(1) $s_0(v) + com(v) \leq s_0(v)^2$.

(2) If $v \xrightarrow{(r)} v'$ by a reduction at depth 0, then $s_0(v') + com(v') < s_0(v) + com(v)$.

The lemma follows from these two. ■

5 Main Results

Now we are in a position to state the main results of this paper. From Theorems 4.3, 4.6 and 4.7, it follows:

Theorem 5.1 (Polystep strong normalization) *For every term t_0 of size s and depth d, the following hold:*

(i) every reduction sequence from t_0 has a length bounded by $O(s^{2^{d+1}})$;

(ii) every term to which t_0 reduces has a size bounded by $O(s^{2^d})$.

Corollary 5.2 (Church-Rosser property) *If t_0 is a term and $t_1 \longleftarrow^* t_0 \longrightarrow^* t_2$, then $t_1 \longrightarrow^* t_3 \longleftarrow^* t_2$ for some term t_3.*

Proof. By showing local confluence, which is straightforward. ■

To make precise what we mean by *polytime* strong normalization, we give the following definitions. A *reduction strategy for* $\mathcal{T}$ is a partial function $f : \mathcal{T} \longrightarrow \{0,1\}^*$ such that $f(t)$ gives an address of a redex of t whenever t is reducible and is undefined otherwise. We can think of a Turing machine $\mathsf{normalize}_f$ with function oracle f, described as follows:

```
input t
loop
   query to oracle f to obtain f(t)
   if f(t) is defined
      then let t := t' such that t --f(t)--> t'
      else output t and halt
end loop.
```

Now we have:

Corollary 5.3 (Polytime strong normalization) *For any reduction strategy f for $\mathcal{T}$,* $\mathsf{normalize}_f$ *terminates in time $O(s^{2^{d+2}})$, given a term t_0 of size s and depth d as input. It outputs the normal form of t_0.*

Proof. Observe that each step of reduction $t \longrightarrow t'$ is carried out in quadratic time: the worst case, namely the case of (!)-reduction, consists in substituting a subterm of size $\leq |t|$ for at most $|t|$ variable occurrences. Therefore the total runtime is roughly estimated by $O(s^{2^{d}\cdot 2} \cdot s^{2^{d+1}}) = O(s^{2^{d+2}})$. ■

Finally let us mention the converse of Theorem 3.8. (This is essentially due to [10, 2], but we include it here for self-containedness.)

Theorem 5.4 *Every term t of type* $\mathbf{bint} \multimap \S^d\mathbf{bint}$ *represents a function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ which is computable in time $O(n^{2^{d+3}})$.*

Proof. Recall that all $\overline{w}$'s are of depth 1, so that $t\overline{w}$ is of constant depth for every $w \in \{0,1\}^*$. Without loss of generality, we may assume that the depth is equal to the depth of $\S^d\mathbf{bint}$, i.e., $d+1$ (just ignore the deeper layers, which do not contribute to the normal form; see Remark 3.7). By Corollary 5.3, the normal form of $t\overline{w}$ is computed in time $O(|t\overline{w}|^{2^{d+3}})$, thus in time $O(|w|^{2^{d+3}})$ (by taking a reasonable reduction strategy of low complexity). The normal form should be of the form $\S^d\overline{w'}$, and such w' is unique by the Church-Rosser property. ■

Corollary 5.5 (Characterization of the Polytime Functions) *A function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ is polytime computable if and only if it is represented by a* λ_{LA} *term of type* $\mathbf{bint} \multimap \S^d\mathbf{bint}$ *for some d.*

Observe, however, that there is an exponential gap between the representability (a function computable in time $O(n^d)$ is representable by a term of depth $d+7$) and the normalizability (a term of depth d is normalizable in time $O(n^{2^{d+2}})$).

6 Concluding Discussion

We have introduced an untyped term calculus λ_{LA}, which has **ILAL** as a type assignment system, and showed the polytime strong normalization theorem for λ_{LA}. It follows that every term typable in **ILAL**, which can be considered as structurally representing an **ILAL** proof, is polytime strongly normalizable.

Strong polytime normalization for LLL. Before turning to **LLL**, let us consider decompositions of the (!) reduction rule:

$(!_1)$ let $!u$ be $!x$ in $\Phi[x] \longrightarrow$ let $!u$ be $!x$ in $\Phi[u]$;

$(!_2)$ let $!u$ be $!x$ in $t \longrightarrow t$, if $x \notin FV(t)$.

Clearly the (!) reduction rule is simulated by these two. With this modification, we still have the polytime strong normalization theorem. Note that these rules are natural counterparts of Girard[10]'s reduction rules for the exponential boxes: $(!_1)$ corresponds to the contraction reduction and $(!_2)$ to the weakening reduction.

Given this, it is quite plausible that we can apply our technique to **LLL** to show the strong polytime normalization theorem for the proofnets of **LLL**(with formulas erased). There is, however, a limitation that additives should be treated in a lazy way, because eager reductions of additive boxes cost exponential time.

On weakly polytime programs and interpretation of safe recursion. Let us consider polytime programs in functional programming in general. Since execution of such programs depends on reduction strategies, it makes sense to classify them into the *strongly polytime* programs (which are polytime executable by any strategy) and the *weakly polytime* ones (which are polytime executable only by some strategy). λ_{LA} accepts only strongly polytime programs. By contrast, most polytime type systems based on *safe recursion* ([7, 16]) accept weakly polytime programs, too (see, e.g., [15, 11, 8]). Typically they allow the following conditional

$$\mathsf{cond}(x) := \mathsf{if}\ p(x)\ \mathsf{then}\ f_1(x)\ \mathsf{else}\ f_2(x)$$

to be iterated when the argument x is *safe*. It is easy to see that iteration of cond is weakly polytime but not strongly, since unfolding the iteration without computing the conditional yields a term of exponential size. (By the way, observe that iteration of this kind of conditionals is the key to encode Turing machine computations: think of p as discriminating the current configuration and f_1 and f_2 as transforming it accordingly. Being strongly polytime systems, light systems do not allow conditionals like above to be iterated, at least in full generality. That is why the encoding of Turing machines is so delicate in light systems (see [19, 3])).

An interesting consequence is that there cannot be a "reasonable" embedding of those type systems of safe recursion into λLA which *preserves the reduction relation*. To be more precise, there is no inductive embedding such that

- it maps numerals of the former systems to λLA terms of polynomial size and of constant depth, and
- whenever t one-step reduces to u in the former systems, the translation of t reduces to that of u in several (but not zero) steps in λLA.

Therefore there is a limitation on the interpretability of safe recursion; although there still remains a possibility to have an non-reduction-preserving embedding which *prunes* exponential reduction sequences in the original system so that a weakly polytime program is transfigured into a strongly polytime one. (This remark is complementary to the result of [17], which shows that *safe recursion with noncontractible safe variables* is interpretable in **ILAL**.)

We leave the following to future work:

- Pursuit of efficiency in normalization. The polynomial time bound given in this paper describes the complexity of the worst reduction strategy among all possible ones. It seems likely that we can significantly improve it by specifying a wiser strategy (perhaps a deeper-layer-first one). In particular we would like to know if it is possible to fill the exponential gap mentioned in the last of the previous section.
- Incorporation of inductive data types as primitives, while keeping the polytime upperbound for normalization; it will make λLA more accessible to programmers.
- Extension of the light logical approach to other complexity classes, such as polynomial hierarchy and polynomial space.

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] A. Asperti. Light affine logic. In *Proceedings of LICS'98*, 1998.

[3] A. Asperti and L. Roversi. Intuitionistic light affine logic (proof-nets, normalization complexity, expressive power, programming notation). Submitted, 2000.

[4] P. Baillot. Stratified coherent spaces: a denotational semantics for light linear logic. Presented at the Second International Workshop on Implicit Computational Complexity, 2000.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier North-Holland, 1981.

[6] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford University Press, 1992.

[7] S. Bellantoni and S. Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[8] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Ramification, modality and linearity in higher type recursion. Presented at the First International Workshop on Implicit Computational Complexity, 1999.

[9] J.-Y. Girard. Light linear logic. manuscript, 1995.

[10] J.-Y. Girard. Light linear logic. *Information and Computation*, 14(3):175–204, 1998.

[11] M. Hofmann. *Type Systems for Polynomial-Time Computation.* Habilitationsschrift, Technical University of Darmstadt, 1998.

[12] M. Kanovitch, M. Okada, and A. Scedrov. Phase semantics for light linear logic. *Theoretical Computer Science*, to appear. An extended abstract appeared in Proceedings of MFPS'97.

[13] D. Leivant. A foundational delineation of poly-time. *Information and Computation*, 1994.

[14] D. Leivant. Applicative control and computational complexity. In *Proceedings of CSL'99*, pages 82–95. Springer-Verlag, LNCS 1683, 1999.

[15] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19:167–184, 1993.

[16] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 320 – 343. Birkhauser, 1994.

[17] A. S. Murawski and C.-H. L. Ong. Can safe recursion be interpreted in light logic? Presented at the Second International Workshop on Implicit Computational Complexity, 2000.

[18] A. S. Murawski and C.-H. L. Ong. Discreet games, light affine logic and ptime computation. In *Proceedings of CSL2000*, pages 427–441. Springer-Verlag, LNCS 1862, 2000.

[19] L. Roversi. A P-time completeness proof for light logics. In *Proceedings of CSL'99*, pages 469 -483. Springer-Verlag, LNCS 1683, 1999.

[20] L. Roversi. Light affine logic as a programming language: a first contribution. *International Journal of Foundations of Computer Science*, 11(1):113 – 152, March 2000.

[21] P. Wadler. A syntax for linear logic. In *Proceedings of MFPS'93*. Springer Verlag, LNCS 802, 1993.

[22] J. B. Wells. Typability and type-checking in the second-order calculus are equivalent and undecidable. In *Proceedings of LICS'94*, pages 176 185. IEEE, 1994.

Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory

Frank Pfenning*
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

Abstract

We develop a uniform type theory that integrates intensionality, extensionality, and proof irrelevance as judgmental concepts. Any object may be treated intensionally (subject only to α-conversion), extensionally (subject also to $\beta\eta$-conversion), or as irrelevant (equal to any other object at the same type), depending on where it occurs. Modal restrictions developed in prior work for simple types are generalized and employed to guarantee consistency between these views of objects. Potential applications are in logical frameworks, functional programming, and the foundations of first-order modal logics.

Our type theory contrasts with previous approaches that a priori distinguish propositions (whose proofs are all identified—only their existence is important) from specifications (whose implementations are subject to some definitional equalities).

1 Introduction

In the development of type theory, there has been considerable debate about the degree of extensionality or intensionality that should be inherent in its formulation. In an extensional theory such as the one underlying Nuprl [4] type-checking is undecidable. In a non-extensional theory[1] such as later versions of Martin-Löf's type theory [17], we distinguish a *definitional equality* (also called *judgmental equality*) which is not extensional and decidable, from a *propositional equality* which is extensional and undecidable. There are a number of tradeoffs, both from the philosophical and pragmatic points of view. In an undecidable, extensional theory, programs are significantly more compact than in a decidable, non-extensional theory. On the other hand, we need external arguments to validate the correctness of programs, defeating at least in part the motivations underlying the separation of judgments from propositions [11, 12]. Furthermore, the development of extensional concepts in a non-extensional type theory is far from straightforward, as can be seen from Hofmann's systematic study [10].

Related is the issue of *proof irrelevance*, which plays an important role in the development of mathematical concepts in type theory via subset types or quotient types. For example, the type $\{x{:}A \mid B(x)\}$ should contain the elements M of type A that satisfy property B. If we want type-checking to be decidable, we require evidence that $B(M)$ is satisfied, but we should not distinguish between different proofs of $B(M)$—they are *irrelevant*.

In this paper we present a type theory that internalizes the concepts of intensionality, extensionality, and proof irrelevance via distinctions familiar from modal logic. We strictly follow Martin-Löf's separation of judgments from propositions and both type-checking and definitional equality are decidable.

At the heart of our modal type theory are three judgments

$$\begin{array}{ll} M :: A & M \text{ is an } \textit{expression} \text{ of type } A, \\ M : A & M \text{ is an } \textit{term} \text{ of type } A, \text{ and} \\ M \div A & M \text{ is a } \textit{proof} \text{ of type } A, \end{array}$$

constructed from the same set of objects M and types A. Expressions are treated *intensionally*: they are subject only to α-conversion. Terms are treated *extensionally*: they are additionally subject to β and η-conversion. Proofs are treated as if *irrelevant*: any two proofs of the same type are identified. All these are part of the definitional equality of the type theory, which therefore combines intensionality, extensionality, and irrelevance into a single system in a coherent way.

It is a critical property of our type theory that the distinction between expressions, terms, and proofs is not made at the time the constituent constants are declared, but at the

*This work was partially supported by NSF Grant CCR-9988281.

[1] Such type theories are often called *intensional*, but this is somewhat misleading since the meaning of objects is still subject to some conversion rules.

0-7695-1281-X/01 $10.00 © 2001 IEEE

time those constants are used. Any type A can be seen as the type of an expression, the type of a term (= a specification), or the type of a proof (= a proposition). Similarly, an object M may be seen as an expression, as a term, or as a proof, depending only on whether some conditions on its free variables are satisfied. We believe that this flexibility is an inherent advantage of our approach compared to a priori separating propositions (inhabited by proofs that are always irrelevant) from specifications (inhabited by terms that are never irrelevant). This is the approach mostly taken in the literature (see, for example, [18] or, allowing even for some classical reasoning, [2]).

Our system is also interesting in its relation to intuitionistic modal logic when we ignore the objects. Our default judgment $M : A$ can be interpreted as "*A is true*". The judgment $M :: A$ can be read as "*A is valid*". The judgment $M \div A$ can be read as "*A is provable*", hiding the proof object. These can be seen as modes of truth, and the work presented here is an extension of prior work on proof term calculi for the modal logic S4 [20] where validity corresponds to necessary truth.

In a type theory as a foundation for functional programming, irrelevant objects (that is, proofs) are erased before execution without affecting the observable outcome. From this point of view, our type system internally captures a notion of dead-code elimination (see, for example, [1] for a survey and position paper on related type-based approaches). However, we need to extend our type theory with first-class modal operators in order to use it in the context of a complete functional language. Two non-dependent theories in this style are given in [20], explaining an intuitionistic modal logic with necessity ($\Box A$) and possibility ($\Diamond A$). A proper treatment of the fully dependent version of these theories would seem to require an equational theory with commuting conversions and is therefore left to future work. Fortunately, it is possible to develop a consistent and useful type theory where these judgments are considered primarily as hypotheses. Instead of internalizing them as modal operators, we internalize the corresponding hypothetical judgment as function types. Such a restriction is not new—it goes back to similar treatments of linear logic [9] and linear type theory [3] with similar motivations.

In the remainder of the paper we present our type theory, investigate its properties, and sketch some further developments and potential applications.

2 A Modal Type Theory

Our modal type theory is a conservative extension of LF [7]. Our approach follows the outline of [8], adapted here to our more general type theory. The interested reader may find additional details in [19].

2.1 Syntax

The syntax is stratified into objects, families, and kinds as for LF.

$$
\begin{array}{rrcl}
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.\ K \\
 & & & \mid \Pi x{::}A.\ K \mid \Pi x{\div}A.\ K \\
\text{Families} & A & ::= & a \mid A\,M \mid \Pi x{:}A_1.\ A_2 \\
 & & & \mid A \bullet M \mid \Pi x{::}A_1.\ A_2 \\
 & & & \mid A \circ M \mid \Pi x{\div}A_1.\ A_2 \\
\text{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\,M_2 \\
 & & & \mid \lambda x{::}A.\ M \mid M_1 \bullet M_2 \\
 & & & \mid \lambda x{\div}A.\ M \mid M_1 \circ M_2 \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, x{::}A \mid \Gamma, x{\div}A
\end{array}
$$

Here, $M_1 \bullet M_2$ is an application whose argument (M_2) is treated as an expression (intensionally), while $M_1 \circ M_2$ is an application whose argument is treated as a proof (irrelevant for equality). We use K for kinds, A, B, C for type families, M, N, P for objects, Γ for contexts and Σ for signatures. We also use the symbol "kind" to classify the valid kinds. We consider terms that differ only in the names of their bound variables as identical. We write $[N/x]M$, $[N/x]A$ and $[N/x]K$ for capture-avoiding substitution. Signatures and contexts may declare each constant and variable at most once. For example, when we write $\Gamma, x{:}A$ we assume that x is not already declared in Γ. If necessary, we tacitly rename x before adding it to the context Γ. Since a signature is generally fixed, and constants may be used anywhere, we have permitted only two forms of constant declaration, namely $a{:}K$ and $c{:}A$. Note that this is not a restriction for our applications, since it is the *use* not the definition of a constant which determines its status with respect to definitional equality.

2.2 Judgments

The modal type theory is defined by the following principal judgments.

$\vdash \Sigma\ \mathsf{sig}$	Σ is a valid signature
$\vdash_\Sigma \Gamma\ \mathsf{ctx}$	Γ is a valid context
$\Gamma \vdash_\Sigma M : A$	M has type A
$\Gamma \vdash_\Sigma A : K$	A has type K
$\Gamma \vdash_\Sigma K : \mathsf{kind}$	K is a valid kind
$\Gamma \vdash_\Sigma M = N : A$	M extensionally equals N
$\Gamma \vdash_\Sigma A = B : K$	A extensionally equals B
$\Gamma \vdash_\Sigma K = L : \mathsf{kind}$	K extensionally equals L
$\Gamma \vdash_\Sigma M \equiv N : A$	M intensionally equals N

As explained later, intensional equality for types and kinds is not needed directly, and proof irrelevance is a derived concept.

For the judgment $\vdash_\Sigma \Gamma$ ctx we presuppose that Σ is a valid signature. For the remaining judgments of the form $\Gamma \vdash_\Sigma J$ we presuppose that Σ is a valid signature and that Γ is valid in Σ. For the sake of brevity we omit the signature Σ from all judgments but the first, since it does not change throughout a derivation.

If J is a typing or equality judgment, then we write $[M/x]J$ for the obvious substitution of M for x in J. For example, if J is $N : B$, then $[M/x]J$ stands for the judgment $[M/x]N : [M/x]B$.

We also have several derived judgments that are central the nature of our type theory. Each of them is defined by only a single rule. In order to explain these additional judgments we need two critical operations on contexts. The first, $\Gamma^\ominus$, hides all term variables $x{:}A$ by converting them to proof variables $x \div A$. The second, $\Gamma^\oplus$, resurrects all proof variables $x \div A$ by converting them to term variables $x{:}A$. Other declarations are unaffected in both cases.

$$\begin{array}{rr} (\cdot)^\ominus = \cdot & (\cdot)^\oplus = \cdot \\ (\Gamma, x{:}A)^\ominus = \Gamma^\ominus, x \div A & (\Gamma, x{:}A)^\oplus = \Gamma^\oplus, x{:}A \\ (\Gamma, x{::}A)^\ominus = \Gamma^\ominus, x{::}A & (\Gamma, x{::}A)^\oplus = \Gamma^\oplus, x{::}A \\ (\Gamma, x \div A)^\ominus = \Gamma^\ominus, x \div A & (\Gamma, x \div A)^\oplus = \Gamma^\oplus, x{:}A \end{array}$$

Intensional Expressions. The new judgments

$$\begin{array}{rl} \Gamma \vdash_\Sigma M :: A & M \text{ is an expression of type } A \\ \Gamma \vdash_\Sigma A :: K & A \text{ is an expression type of kind } K \\ \Gamma \vdash_\Sigma M = N :: A & M \text{ and } N \text{ are equal expressions} \\ \Gamma \vdash_\Sigma A = B :: K & A \text{ and } B \text{ are equal expression types} \end{array}$$

are defined by the following rules

$$\frac{\Gamma^\ominus \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma M :: A} \qquad \frac{\Gamma^\ominus \vdash_\Sigma A : K}{\Gamma \vdash_\Sigma A :: K}$$

$$\frac{\Gamma^\ominus \vdash_\Sigma M \equiv N : A}{\Gamma \vdash_\Sigma M = N :: A} \qquad \frac{\Gamma^\ominus \vdash_\Sigma A = B : K}{\Gamma \vdash_\Sigma A = B :: K}$$

The idea is that an expression cannot refer to a term variable $x{:}B$, which would violate intensionality. Thus we mark these variables as irrelevant, $x \div B$, which is accomplished by the $(\,)^\ominus$ operation. Note, however, that intensionality and irrelevance interact: proof variables may still occur in an intensional expression, but only inside other proofs! The rules for equality indicate that only intensionally equal terms are considered as equal expressions. We do not directly refer to α-convertibility here because expressions may contain proofs that must be identified, even as subterms of expressions. Note that expression types are not intensional, but that there is a restriction regarding their validity: expression types can not depend on term variables directly.

In general, $M :: A$ is inherently stronger than $M : A$, that is, $M :: A$ implies $M : A$ but not vice versa. In particular, $x{:}A \not\vdash_\Sigma x :: A$.

Irrelevant Proofs. The new judgments

$$\begin{array}{ll} \Gamma \vdash_\Sigma M \div A & M \text{ is a proof of type } A \\ \Gamma \vdash_\Sigma A \div K & A \text{ is a proof type of kind } K \\ \Gamma \vdash_\Sigma M = N \div A & M \text{ and } N \text{ are equal proofs} \\ \Gamma \vdash_\Sigma A = B \div K & A \text{ and } B \text{ are equal proof types} \end{array}$$

are defined by the following rules

$$\frac{\Gamma^\oplus \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma M \div A} \qquad \frac{\Gamma^\oplus \vdash_\Sigma A : K}{\Gamma \vdash_\Sigma A \div K}$$

$$\frac{\Gamma^\oplus \vdash_\Sigma M = M : A \qquad \Gamma^\oplus \vdash_\Sigma N = N : A}{\Gamma \vdash_\Sigma M = N \div A}$$

$$\frac{\Gamma^\oplus \vdash_\Sigma A = B : K}{\Gamma \vdash_\Sigma A = B \div K}$$

The idea is that a proof may depend on expression variables, term variables, and proof variables. This effect is achieved by relabelling hypotheses $x \div B$ to $x{:}B$ in the $(\,)^\oplus$ operation. Note that equality between proofs implements *proof irrelevance* in the classical sense. We could replace the premise $\Gamma^\oplus \vdash_\Sigma M = M : A$ with $\Gamma^\oplus \vdash_\Sigma M : A$ (and similarly for N), but for technical reasons it is simpler if the equality judgment does not refer to the typing judgment here.

It is important that $M \div A$ is inherently weaker than $M : A$. In particular, $x \div A \not\vdash_\Sigma x : A$. In other words, terms can not depend on proof variables, but other proofs can. Under a functional interpretation, it is this property which allows the consistent erasure of all proof objects without affecting the observable outcome (assuming proofs are not observable).

Note that, unlike the systems in [5, 20], the rules have the property of *variable monotonicity*: when viewed bottom-up, every variable is preserved—only its status might change from the conclusion to the premise of a rule. This is inspired by a similar idea in [13] and is needed for a clean interaction between expressions and proofs.

2.3 Typing Rules

Our formulation of the typing rules is similar to the second version given in [7] and directly based on [8]. In preparation for the various algorithms we presuppose and inductively preserve the validity of contexts involved in the judgments, instead of checking these properties at the leaves. This is a matter of expediency rather than necessity. Furthermore, in order to the shorten the presentation we use the following notation:

> "$\star$" stands for either ":", "::", or "$\div$" were all occurrences in a rule must be consistent.

Objects.

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash c : A} \qquad \frac{}{\Gamma, x{:}A, \Gamma' \vdash x : A} \qquad \frac{}{\Gamma, x{::}A, \Gamma' \vdash x : A} \quad \textit{no rule for } x \div A$$

$$\frac{\Gamma \vdash A_1 \star \text{type} \qquad \Gamma, x \star A_1 \vdash M_2 : A_2}{\Gamma \vdash \lambda x \star A_1.\ M_2 : \Pi x \star A_1.\ A_2} \qquad \frac{\Gamma \vdash M_1 : \Pi x \star A_2.\ A_1 \qquad \Gamma \vdash M_2 \star A_2}{\Gamma \vdash M_1 * M_2 : [M_2/x]A_1}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A = B : \text{type}}{\Gamma \vdash M : B}$$

Families.

$$\frac{a{:}K \text{ in } \Sigma}{\Gamma \vdash a : K} \qquad \frac{\Gamma \vdash A : \Pi x \star B.\ K \qquad \Gamma \vdash M \star B}{\Gamma \vdash A * M : [M/x]K}$$

$$\frac{\Gamma \vdash A_1 \star \text{type} \qquad \Gamma, x \star A_1 \vdash A_2 : \text{type}}{\Gamma \vdash \Pi x \star A_1.\ A_2 : \text{type}} \qquad \frac{\Gamma \vdash A : K \qquad \Gamma \vdash K = L : \text{kind}}{\Gamma \vdash A : L}$$

Figure 1. Rules for Validity of Objects and Families

"$*$" stands for either juxtaposition (an application of a function of type $\Pi x{:}A.\ B$), "$\bullet$" (an application of a function of type $\Pi x{::}A.\ B$), or "$\circ$" (an application of a function of type $\Pi x \div A.\ B$). Occurrences of "$*$" must be coordinated with occurrences of "$\star$" in a rule schema in the indicated manner.

Signatures. The rules for validity of signatures are straightforward and omitted here. From now on we fix a valid signature Σ and omit it from the judgments.

Contexts. Validity of contexts must guarantee that we cannot incorrectly refer to a proof variable in a term or expression, or a term variable in an expression. This is achieved by the following rules.

$$\frac{}{\vdash \cdot \text{ ctx}} \qquad \frac{\vdash \Gamma \text{ ctx} \qquad \Gamma \vdash A \star \text{type}}{\vdash \Gamma, x \star A \text{ ctx}}$$

Note that the second rule schema actually stands for three rules, depending on whether $x{:}A$, $x{::}A$, or $x \div A$ appear in the conclusion and premise.

Objects. Here we proceed as in LF, except that we need to make sure that arguments fit the type and disposition (intensional, extensional, or irrelevant) of the function. The rules can be found in Figure 1. The rule schema for application is the most complex and has three instances. One of them, for example, replaces $\star$ by :: and $*$ by $\bullet$.

Families and Kinds. The rules for application and conversion are copies of the rules from the level of objects. Valid function types restrict occurrences of the dependent variable based on whether the corresponding argument is interpreted as an expression, a term, or a proof. This is necessary to guarantee that the type of an application, which is obtained by substitution, is valid. The rules at the level of kinds mirror the ones at the level of families and are elided here.

Generally, in our theory the judgments on families only reflect the judgments on the objects embedded in them. This is typical of type theories such as the one underlying LF.

2.4 Definitional Equality

The rules for definitional are written with the presupposition that a valid signature Σ is fixed and that all contexts Γ are valid. The intent is that equality implies validity of the objects, families, or kinds involved (see Lemma 2). In contrast to the original formulation of LF in [7], equality of terms is based on a notion of parallel conversion plus extensionality, rather than $\beta\eta$-conversion, but the two definitions turn out to be equivalent. In addition we have to take care of intensionality for expressions and irrelevance of proofs. This is reflected in the rules for intensional application $M \bullet N$ and irrelevant application $M \circ N$.

Some of the typing premises in the rules are redundant, but for technical reasons we cannot prove this until validity has been established. Such premises are enclosed in {braces}.

Simultaneous Congruence.

$$\frac{c{:}A \text{ in } \Sigma}{\Gamma \vdash c = c : A} \qquad \frac{}{\Gamma, x{:}A, \Gamma' \vdash x = x : A} \qquad \frac{}{\Gamma, x{::}A, \Gamma' \vdash x = x : A}$$

$$\frac{\Gamma \vdash M_1 = N_1 : \Pi x{\star}A_2.\ A_1 \qquad \Gamma \vdash M_2 = N_2 \star A_2}{\Gamma \vdash M_1 * M_2 = N_1 * N_2 : [M_2/x]A_1}$$

$$\frac{\Gamma \vdash A_1' = A_1 \star \text{type} \qquad \Gamma \vdash A_1'' = A_1 \star \text{type} \qquad \Gamma, x{\star}A_1 \vdash M_2 = N_2 : A_2}{\Gamma \vdash \lambda x{\star}A_1'.\ M_2 = \lambda x{\star}A_1''.\ N_2 : \Pi x{\star}A_1.\ A_2}$$

Extensionality.

$$\frac{\Gamma \vdash A_1 \star \text{type} \quad \{\Gamma \vdash M : \Pi x{\star}A_1.\ A_2\} \quad \{\Gamma \vdash N : \Pi x{\star}A_1.\ A_2\} \quad \Gamma, x{\star}A_1 \vdash M * x = N * x : A_2}{\Gamma \vdash M = N : \Pi x{\star}A_1.\ A_2}$$

Parallel Reduction.

$$\frac{\{\Gamma \vdash A_1 \star \text{type}\} \qquad \Gamma, x{\star}A_1 \vdash M_2 = N_2 : A_2 \qquad \Gamma \vdash M_1 = N_1 \star A_1}{\Gamma \vdash (\lambda x{\star}A_1.\ M_2) * M_1 = [N_1/x]N_2 : [M_1/x]A_2}$$

Type Conversion.

$$\frac{\Gamma \vdash M = N : A \qquad \Gamma \vdash A = B : \text{type}}{\Gamma \vdash M = N : B}$$

Figure 2. Extensional Equality Between Objects

Objects. The extensional equality rules for objects are shown in Figure 2, where we have elided rules stating symmetry and transitivity. Conversion is modelled by parallel reduction, a choice motivated by technical concerns. Reflexivity is admissible, which is typical for equality based on parallel reduction.

The crux of intensionality and irrelevance is in the cases for the corresponding applications, $M \bullet N$ and $M \circ N$. We therefore explicitly consider the second premise in the rule schema for application in its three specific instances.

If we compare $M_1\, M_2 = N_1\, N_2$, then the second premise requires $M_2 = N_2 : A_2$, just as in LF.

If we compare $M_1 \bullet M_2 = N_1 \bullet N_2$ then the arguments are treated intensionally and equality will only succeed if M_2 and N_2 are well-typed and intensionally equal expressions. This is enforced with the judgment $\Gamma \vdash M_2 = N_2 :: A_2$ defined before, which holds if and only if $\Gamma^{\ominus} \vdash M_2 \equiv N_2 : A_2$.

If we compare $M_1 \circ M_2 = N_1 \circ N_2$ then the arguments are proofs and are always considered equal. We only need to check that they are well-typed, which is accomplished with the judgment $\Gamma \vdash M_2 = N_2 \div A_2$ defined before. This holds if and only if $\Gamma^{\oplus} \vdash M_2 : A_2$ and $\Gamma^{\oplus} \vdash N_2 : A_2$.

Since the main equality judgment compares terms and not expressions or proofs, the extensionality principle holds for all three kinds of functions. Modulo the construction of the right kind of context and some redundant premises required for technical reasons, these are straightforward. Similarly, the rule of parallel reduction is available for all three kinds of functions.

Families and Kinds. The rules in Figure 2 are repeated with straightforward adaptations at the levels of families and kinds and omitted here. Details can be found in the technical report [19].

Intensional Equality. The intensional equality between objects, $\Gamma \vdash M \equiv N : A$, is defined as a simultaneous congruence just as the extensional equality, but we delete the rules for extensionality and parallel conversion. In the modified rules, arguments to functions that are to be treated as proofs, however, are considered irrelevant for equality as before. Hence irrelevance takes precedence over intensionality, which seems most appropriate for the intended applications as outlined in Section 7. The reader can find the full set of rules in [19].

2.5 Elementary Properties

We establish some elementary properties of the judgments pertaining to the interpretation of contexts. All of these have standard or straightforward proofs on the structure of derivations. First we show weakening for all judgments of the type theory. Secondly, reflexivity holds for valid objects, families, and kinds.

For all lemmas and theorems from here on we tacitly assume that the contexts in the given derivations are well-formed. Furthermore, in the statement of a meta-theoretic property, several occurrences of "$\star$" must still be instantiated consistently as for inference rules.

Lemma 1 (Substitution) *If* $\Gamma, x{\star}A, \Gamma' \vdash J$ *and* $\Gamma \vdash M \star A$ *then* $\Gamma, [M/x]\Gamma' \vdash [M/x]J$.

Proof: By induction over the structure of the first given derivation. □

Note that this is shorthand for several separate substitution properties. Now there is a series of technical lemmas (which we omit), culminating in validity and functionality.

Lemma 2 (Validity)

1. *If* $\Gamma \vdash M \star A$ *then* $\Gamma \vdash A \star$ type.
2. *If* $\Gamma \vdash M = N \star A$, *then* $\Gamma \vdash M \star A$, $\Gamma \vdash N \star A$, *and* $\Gamma \vdash A \star$ type.

Analogous properties hold at the levels of families and kinds.

Lemma 3 (Functionality) *If* $\Gamma \vdash M = N \star A$ *and* $\Gamma, x{\star}A \vdash O = P : B$ *then* $\Gamma \vdash [M/x]O = [N/x]P : [M/x]B$ *and similarly at the level of types and kinds.*

Another consequence of validity is a collection of standard inversion properties. In the interest of space, we elide these properties here. We can further show, from validity, that the premises enclosed in $\{\ldots\}$ are indeed redundant, that is, follow from the other premises.

3 An Algorithm for Deciding Equality

The algorithm for deciding definitional equality can be summarized as follows:

1. When comparing objects at function type, apply extensionality.
2. When comparing objects at base type, reduce both sides to weak head-normal form and then compare heads directly. If they are equal, we compare each corresponding pair of arguments according to their status.
   (a) When the corresponding arguments are extensional (terms), recursively compare for extensional equality.
   (b) When the corresponding arguments are intensional (expressions), compare for syntactic equality modulo α-conversion, ignoring only embedded proof terms.
   (c) When the corresponding arguments are irrelevant (proofs), we always treat them as equal.

Since this algorithm is type-directed in case (1) we need to carry types. Unfortunately, this makes it difficult to prove correctness of the algorithm in the presence of dependent types, because transitivity is not an obvious property. Fortunately, we do not need to know the precise type of the objects we are comparing.

We therefore define a calculus of simple approximate types and an erasure function $()^-$ that eliminates dependencies for the purpose of this algorithm. Note that there are three forms of non-dependent function type which we write as $\tau_1 \stackrel{\star}{\rightarrow} \tau_2$ and similarly for kinds.

We write α to stands for simple base types and we have two special type constants, type$^-$ and kind$^-$, for the equality judgments at the level of types and kinds.

$$\begin{array}{rl} \text{Simple Kinds} & \kappa ::= \text{type}^- \mid \tau \stackrel{\cdot}{\rightarrow} \kappa \mid \tau \stackrel{::}{\rightarrow} \kappa \mid \tau \stackrel{\div}{\rightarrow} \kappa \\ \text{Simple Types} & \tau ::= \alpha \mid \tau_1 \stackrel{\cdot}{\rightarrow} \tau_2 \mid \tau_1 \stackrel{::}{\rightarrow} \tau_2 \mid \tau_1 \stackrel{\div}{\rightarrow} \tau_2 \\ \text{Simple Contexts} & \Delta ::= \cdot \mid \Delta, x{:}\tau \mid \Delta, x{::}\tau \mid \Delta, x{\div}\tau \end{array}$$

We use τ, θ, δ for simple types and Δ for contexts declaring simple types for variables. We also use "kind$^-$" in a similar role to "kind" in the LF type theory.

We write A^- for the simple type that results from erasing dependencies in A, and similarly K^-. We translate each constant type family a to a base type a^- and extend this to all type families. We extend it further to contexts by applying it to each declaration.

$$\begin{array}{rcl} (a)^- & = & a^- \\ (A * M)^- & = & A^- \\ (\Pi x{\star}A_1.\ A_2)^- & = & A_1^- \stackrel{\star}{\rightarrow} A_2^- \end{array}$$

We now present the algorithm in the form of four judgments. These can be interpreted as an algorithm in the manner of logic programming.

$M \stackrel{\text{whr}}{\longrightarrow} M'$ (M *weak head reduces to* M') Algorithmically, we assume M is given and compute M' (if M is head reducible) or fail.

$\Delta \vdash M \Longleftrightarrow N : \tau$ (M *is equal to* N *at simple type* τ) Algorithmically, we assume Δ, M, N, and τ are given and we simply succeed or fail. We only apply this judgment if M and N have the same type A and $\tau = A^-$.

$\Delta \vdash M \longleftrightarrow N : \tau$ (*M is structurally equal to N*) Algorithmically, we assume that Δ, M and N are given and we compute τ or fail. If successful, τ will be the approximate type of M and N.

$\Delta \vdash M \Leftrightarrows N$ (*M is intensionally equal to N*) Algorithmically, we assume that Δ, M, and N are given and we either succeed or fail.

Note that the structural and type-directed equality are mutually recursive, while weak head reduction does not depend on the other three judgments.

Weak Head Reduction.

$$\frac{}{(\lambda x{\star}A_1.\ M_2) * M_1 \xrightarrow{\text{whr}} [M_1/x]M_2}$$

$$\frac{M_1 \xrightarrow{\text{whr}} M_1'}{M_1 * M_2 \xrightarrow{\text{whr}} M_1' * M_2}$$

Type-Directed Object Equality.

$$\frac{M \xrightarrow{\text{whr}} M' \qquad \Delta \vdash M' \Longleftrightarrow N : \alpha}{\Delta \vdash M \Longleftrightarrow N : \alpha}$$

$$\frac{N \xrightarrow{\text{whr}} N' \qquad \Delta \vdash M \Longleftrightarrow N' : \alpha}{\Delta \vdash M \Longleftrightarrow N : \alpha}$$

$$\frac{\Delta \vdash M \longleftrightarrow N : \alpha}{\Delta \vdash M \Longleftrightarrow N : \alpha}$$

$$\frac{\Delta, x{\star}\tau_1 \vdash M * x \Longleftrightarrow N * x : \tau_2}{\Delta \vdash M \Longleftrightarrow N : \tau_1 \xrightarrow{\star} \tau_2}$$

Structural Object Equality.

$$\frac{c{:}A \text{ in } \Sigma}{\Delta \vdash c \longleftrightarrow c : A^-} \qquad \frac{x{:}\tau \text{ or } x{::}\tau \text{ in } \Delta}{\Delta \vdash x \longleftrightarrow x : \tau}$$

$$\frac{\Delta \vdash M_1 \longleftrightarrow N_1 : \tau_2 \xrightarrow{\cdot} \tau_1 \qquad \Delta \vdash M_2 \Longleftrightarrow N_2 : \tau_2}{\Delta \vdash M_1\, M_2 \longleftrightarrow N_1\, N_2 : \tau_1}$$

$$\frac{\Delta \vdash M_1 \longleftrightarrow N_1 : \tau_2 \xrightarrow{\cdot\cdot} \tau_1 \qquad \Delta \vdash M_2 \Leftrightarrows N_2}{\Delta \vdash M_1 \bullet M_2 \longleftrightarrow N_1 \bullet N_2 : \tau_1}$$

$$\frac{\Delta \vdash M_1 \longleftrightarrow N_1 : \tau_2 \xrightarrow{\cdot} \tau_1}{\Delta \vdash M_1 \circ M_2 \longleftrightarrow N_1 \circ N_2 : \tau_1}$$

Structural Intensional Object Equality.

$$\frac{c{:}A \text{ in } \Sigma}{\Delta \vdash c \Leftrightarrows c} \qquad \frac{x{:}\tau \text{ or } x{::}\tau \text{ in } \Delta}{\Delta \vdash x \Leftrightarrow x}$$

$$\frac{\Delta \vdash A \Longleftrightarrow B : \text{type}^- \qquad \Delta, x{\star}A^- \vdash M \Leftrightarrows N}{\Delta \vdash \lambda x{\star}A.\ M \Leftrightarrows \lambda x{\star}B.\ N}$$

$$\frac{\Delta \vdash M_1 \Leftrightarrows N_1 \qquad \Delta \vdash M_2 \Leftrightarrows N_2}{\Delta \vdash M_1\, M_2 \Leftrightarrows N_1\, N_2}$$

$$\frac{\Delta \vdash M_1 \Leftrightarrows N_1 \qquad \Delta \vdash M_2 \Leftrightarrows N_2}{\Delta \vdash M_1 \bullet M_2 \Leftrightarrows N_1 \bullet N_2}$$

$$\frac{\Delta \vdash M_1 \Leftrightarrows N_1}{\Delta \vdash M_1 \circ M_2 \Leftrightarrows N_1 \circ N_2}$$

The crux of the definitions above are the rules for structural equality for applications. We omit the corresponding rules at the level of families. Briefly, kind-directed equality simple decomposes Π-types, while structural type equality reprises the rules for structural object equality above.

The algorithmic equality judgments satisfy some straightforward structural properties, including weakening. Furthermore, the algorithm is essentially deterministic in the sense that when comparing terms at base type we have to weakly head-normalize both sides and compare the results structurally. This is because terms that are weakly head reducible will never be considered structurally equal. This property, as well as the symmetry and transitivity of the algorithm are completely straightforward.

4 Completeness of the Equality Algorithm

In this section we summarize the completeness theorem for the type-directed equality algorithm. That is, if two terms are definitionally equal, the algorithm will succeed. The central idea is to proceed by an argument via logical relations defined inductively on the approximate type of an object, where the approximate type arises from erasing all dependencies.

The completeness direction of the correctness proof for type-directed equality states:

$$\text{If } \Gamma \vdash M = N : A \text{ then } \Gamma^- \vdash M \Longleftrightarrow N : A^-.$$

One would like to prove this by induction on the structure of the derivation for the given equality. However, such a proof attempt fails at the case for application. Instead we define a logical relation $\Delta \vdash M = N \in [\![\tau]\!]$ that provides a stronger induction hypothesis so that both

1. if $\Gamma \vdash M = N : A$ then $\Gamma^- \vdash M = N \in [\![A^-]\!]$, and
2. if $\Gamma^- \vdash M = N \in [\![A^-]\!]$ then $\Gamma^- \vdash M \Longleftrightarrow N \in A^-$,

can be proved.

The development can be found in [19], following [8] quite closely, so we omit it here in the interest of brevity.

Theorem 4 (Completeness of the Equality Algorithm) *If* $\Gamma \vdash M = N : A$ *then* $\Gamma^- \vdash M \Longleftrightarrow N : A^-$. *Furthermore, an analogous property holds at the level of families.*

5 Soundness of the Equality Algorithm

In general, the algorithm for type-directed equality is not sound. However, when applied to valid objects of the same type, it is sound and relates only equal terms. This direction requires a number of syntactic lemmas from Section 2.5, but is otherwise mostly straightforward.

Lemma 5 (Subject Reduction) *If* $M \xrightarrow{\text{whr}} M'$ *and* $\Gamma \vdash M : A$ *then* $\Gamma \vdash M' : A$ *and* $\Gamma \vdash M = M' : A$.

Proof: By induction on the definition of weak head reduction, making use of inversion and substitution properties. □

For the soundness of the equality algorithm we need subject reduction and validity (Lemma 2).

Theorem 6 (Soundness of the Equality Algorithm)

1. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *and* $\Gamma^- \vdash M \Longleftrightarrow N : A^-$, *then* $\Gamma \vdash M = N : A$.
2. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : B$ *and* $\Gamma^- \vdash M \longleftrightarrow N : \tau$, *then* $\Gamma \vdash M = N : A$, $\Gamma \vdash A = B$: type *and* $A^- = B^- = \tau$.
3. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : B$ *and* $\Gamma^- \vdash M \mathrel{(\equiv)} N$ *then* $\Gamma \vdash A = B$: type *and* $\Gamma \vdash M \equiv N : A$.

Analogous properties hold for types and kinds.

Proof: By induction on the structure of the given derivations for algorithmic equality, using validity and inversion on the typing derivations. □

6 Decidability

We can now show that the judgments for the equality algorithm constitute a decision procedure on valid terms of the same type. This result is then lifted to yield decidability of all judgments in the type theory. This part of the development is relatively standard. An exposition of the necessary auxiliary judgments and lemmas can be found in [19]. We only show the final result.

Theorem 7 (Decidability)

1. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$ *then it is decidable whether* $\Gamma \vdash M = N : A$.
2. *Given a valid* Γ, M, *and* A, *it is decidable whether* $\Gamma \vdash M : A$.

Corresponding properties hold at the level of families and kinds and for other equality judgments.

We also have that our type theory is conservative over LF. This is important for logical framework applications, since previously established adequacy theorems for encodings will continue to hold in the modal framework.

7 Further Developments and Potential Applications

In this section we consider various possible further developments and potential applications of our ideas.

7.1 Logical Frameworks

The addition of intensional expressions and irrelevant proofs to the logical framework may leads to more direct and more compact encodings in a number of examples.

First, the intensional nature of expressions constitutes a weak form of reflection: arbitrary LF terms are accessible in LF without regard to $\beta\eta$-conversion. At present we do not have any concrete applications for this added expressive power—the primary application of intensional expressions we have in mind is in the richer setting of functional programming explained in Section 7.2 below.

Second, the irrelevant nature of proofs can be used to encode similar situations in object theories, which is quite frequent. For example, in an encoding of linear functions in LF we often have to deal with pairs consisting of the actual function and the proof certifying its linearity. The nature of this proof is, however, irrelevant, as long as it exists. An encoding of this kind might look as shown below. Here we use $A \to B$ for $\Pi x{:}A.\ B$ where x does not occur in B.

$$
\begin{array}{rcl}
\mathit{rawterm} & : & \text{type} \\
\mathit{lam} & : & (\mathit{rawterm} \to \mathit{rawterm}) \to \mathit{rawterm} \\
\mathit{app} & : & \mathit{rawterm} \to \mathit{rawterm} \to \mathit{rawterm} \\
\mathit{linear} & : & \mathit{rawterm} \to \text{type} \\
\cdots & & \\
\mathit{linterm} & : & \Pi E{:}\mathit{rawterm}.\ \Pi L \div \mathit{linear}\ E.\ \text{type}
\end{array}
$$

The definitional equality at type *linterm* now ignores the proofs that the expressions E are indeed linear. A similar situation arises in the encoding of object languages with subtyping, where often all proofs of subtype relationships should be considered equal. The logic programming interpretation of such encodings can go from infeasible to practical if all choice points are discarded after the first proof has

been found. Such an optimization is justified by our modal type theory without any loss of soundness or completeness.

Moreover, the Twelf system [21] can verify automatically that type families (such as *linear* or one implementing object-language subtyping) are in fact decidable using mode and termination analysis [22]. If we agree that irrelevant objects need not be shown in the user interface, then the proofs of type *linear* E that occur in linear terms actually do not need to be represented at all, leading to a potentially significant space savings that may be critical in applications such as proof-carrying code [14] and certifying decision procedures [23]. Another situation in which an implementation may mark objects as irrelevant is if they are uniquely determined, either for syntactic [15] or semantic [16] reasons. While our modal analysis does not cover all of these optimizations, it generalizes some of the core ideas from a fragment of LF to the full type theory.

7.2 Functional Programming

Our given type theory is fully adequate as a logical framework, but clearly not expressive enough to develop verified functional programs as in various implementations of type theory such as Nuprl [4] or Coq [6]. Besides standard constructs such as inductive types or Σ-types that are orthogonal to our considerations, we need to internalize expressions and proofs as modal operators, rather than just arguments to functions. The blueprint for such an integration for expressions has been given in prior work [5, 20], the correct notion of definitional equality in the presence of dependencies was the main missing ingredient. The presence of both expressions and proofs allows a new twist. We show the formation and introduction rules for the corresponding modal operators, expanding the derived judgments:

$$\frac{\Gamma^{\ominus} \vdash A : \text{type}}{\Gamma \vdash \Box A : \text{type}} \qquad \frac{\Gamma^{\ominus} \vdash M : A}{\Gamma \vdash \text{box}\, M : \Box A}$$

$$\frac{\Gamma^{\oplus} \vdash A : \text{type}}{\Gamma \vdash \triangle A : \text{type}} \qquad \frac{\Gamma^{\oplus} \vdash M : A}{\Gamma \vdash \text{tri}\, M : \triangle A}$$

The elimination rules (especially for the $\triangle$ modality) are unfortunately quite complex. To give the idea: we can now represent, for example, the subset type as a proof-irrelevant version of the the strong sum.

$$\{x{:}A \mid B\} \quad = \quad \Sigma x{:}A.\ \triangle B$$

The triangle operator appears to serve the same purpose as the squash type in [10], except here it derived directly from the judgmental level rather than from identity types.

If our operational interpretation of type theory is based on staged computation [5], then the $\triangle$ modality is necessary to reason about staged programs. Besides a natural symmetry between intensionality and irrelevance as extreme forms of decidable equality, this has been our main motivation for developing a type theory that simultaneously supports these concepts. As an example, consider the specification of a staged power function (presuming a type *nat* and a propositional equality $\doteq$):

$$\not\vdash \Pi n{:}nat.\ \Box(\Pi b{:}nat.\ \Sigma m{:}nat.\ m \doteq b^n) : \text{type}$$

This not well-formed because the term variable n is not available in the expression underneath the $\Box$ constructor. This problem is neatly solved with the $\triangle$ modality as follows:

$$\vdash \Pi n{:}nat.\ \Box(\Pi b{:}nat.\ \Sigma m{:}nat.\ \triangle(m \doteq b^n)) : \text{type}$$

This further specifies that the correctness proof for the staged power function may be erased before execution since it is computationally irrelevant.

7.3 First-Order Intuitionistic Modal Logic

If we consider the first-order fragment of our type theory, the three forms of Π-abstraction correspond to three forms of universal quantification. In terms of a Kripke semantics with varying domains, $\Pi x{:}A.\ B$ quantifies over the elements of the current domain only. This means, for example, that $\Pi x{:}A.\ \Box P(x)$ is only well-formed if P has kind $\Pi x{\div}A.\ \text{type}$, because otherwise the truth of $P(x)$ may need to be investigated in worlds in which x does not exist. Yet it is still possible that x occurs, even if P can only talk about elements of the current world, as in $\Pi x{:}A.\ P(x) \to \Box\triangle P(x)$ (which is true, incidentally). The quantifier $\Pi x{::}A.\ B$ quantifies over elements existing in all domains and thus, in general, fewer than $\Pi x{:}A.\ B$. Finally, $\Pi x{\div}A.\ B$ quantifies over all elements of the current world and also elements that existed in some past world. Thus our approach has the potential to shed new light on old debates by allowing various interpretations of quantification to coexist peacefully in a single modal logic.

8 Conclusion

We have presented a dependent type theory that integrates intensionality, extensionality, and proof irrelevance as judgmental notions, based on considerations from modal logic. We proved that equality and type-checking are decidable on the fragment presented here and sketched some possible applications.

The most pressing item of future work is the inclusion of first-class modal operators important for applications in functional programming. The most difficult question here is the right notion of the "default" equality on terms. In

this paper, the term equality was fully *extensional*; for functional programming applications, this will not be tenable and must be replaced by a decidable judgmental equality that is sound with respect to the operational semantics. We conjecture that this can be done without upsetting the "extreme" equalities of expressions and proofs for which there appears to be little leeway. Furthermore, some type theoretic constructs such as universes may require generalizations of our proof techniques.

Acknowledgments. We would like to thank the anonymous referees for various helpful comments and suggestions.

References

[1] S. Berardi, M. Coppo, F. Damiani, and P. Giannini. Type-based useless-code elimination for functional programs. In W. Taha, editor, *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, pages 172–189, Montreal, Canada, Sept. 2000. Springer-Verlag LNCS 1924.

[2] U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 2001. To appear.

[3] I. Cervesato and F. Pfenning. A linear logical framework. *Information and Computation*, 1998. To appear in a special issue with invited papers from LICS'96, E. Clarke, editor.

[4] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 2000. To appear. Preliminary version available as Technical Report CMU-CS-99-153, August 1999.

[6] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[8] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[9] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[10] M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1995. Available as Technical Report CST-117-95.

[11] P. Martin-Löf. Analytic and synthetic judgements in type theory. In P. Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

[12] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[13] A. Momigliano. *Elimination of Negation in a Logical Framework*. PhD thesis, Department of Philosophy, Carnegie Mellon University, Aug. 2000. Available as Technical Report CMU-CS-00-175.

[14] G. C. Necula. Proof-carrying code. In N. D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, Jan. 1997. ACM Press.

[15] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[16] G. C. Necula and S. Rahul. Oracle-based checking of untrusted software. In H. R. Nielson, editor, *Conference Record of the 28th Annual Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, London, England, Jan. 2001. ACM Press.

[17] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[18] C. Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions*. PhD thesis, Université Paris VII, Jan. 1989.

[19] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. Technical Report CMU-CS-01-116, Department of Computer Science, Carnegie Mellon University, Apr. 2001.

[20] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11, 2001. To appear. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[21] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[22] E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, Apr. 1996. Springer-Verlag LNCS 1058.

[23] A. Stump and D. L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

Dependent Types for Program Termination Verification *

Hongwei Xi
University of Cincinnati
hwxi@ececs.uc.edu

Abstract

Program termination verification is a challenging research subject of significant practical importance. While there is already a rich body of literature on this subject, it is still undeniably a difficult task to design a termination checker for a realistic programming language that supports general recursion. In this paper, we present an approach to program termination verification that makes use of a form of dependent types developed in Dependent ML (DML), demonstrating a novel application of such dependent types to establishing a liveness property. We design a type system that enables the programmer to supply metrics for verifying program termination and prove that every well-typed program in this type system is terminating. We also provide realistic examples, which are all verified in a prototype implementation, to support the effectiveness of our approach to program termination verification as well as its unobtrusiveness to programming. The main contribution of the paper lies in the design of an approach to program termination verification that smoothly combines types with metrics, yielding a type system capable of guaranteeing program termination that supports a general form of recursion (including mutual recursion), higher-order functions, algebraic datatypes, and polymorphism.

1 Introduction

Programming is notoriously error-prone. As a consequence, a great number of approaches have been developed to facilitate program error detection. In practice, the programmer often knows certain program properties that must hold in a *correct* implementation; it is therefore an indication of program errors if the actual implementation violates some of these properties. For instance, various type systems have been designed to detect program errors that cause violations of the supported type disciplines.

It is common in practice that the programmer often knows for some reasons that a particular program should terminate if implemented correctly. This immediately implies that a termination checker can be of great value for detecting program errors that cause nonterminating program execution. However, termination checking in a realistic programming language that supports general recursion is often prohibitively expensive given that (a) program termination in such a language is in general undecidable, (b) termination checking often requires interactive theorem proving that can be too involved for the programmer, (c) a minor change in a program can readily demand a renewed effort in termination checking, and (d) a large number of changes are likely to be made in a program development cycle. In order to design a termination checker for practical use, these issues must be properly addressed.

There is already a rich literature on termination verification. Most approaches to automated termination proofs for either programs or term rewriting systems (TRSs) use various heuristics, some of which can be highly involved, to synthesize well-founded orderings (e.g., various path orderings [3], polynomial interpretation [1], etc.). While these approaches are mainly developed for first-order languages, the work in higher-order settings can also be found (e.g., [7]). When a program, which should be terminating if implemented correctly, cannot be proven terminating, it is often difficult for the programmer to determine whether this is caused by a program error or by the limitation of the heuristics involved. Therefore, such automated approaches are likely to offer little help in detecting program errors that cause nonterminating program execution. In addition, automated approaches often have difficulty handling realistic (not necessarily large) programs.

The programmer can also prove program termination in various (interactive) theorem proving systems such as NuPrl [2], Coq [4], Isabelle [8] and PVS [9]. This is a viable practice and various successes have been reported. However, the main problem with this practice is that the programmer may often need to spend so much time on proving the termination of a program compared with the time spent on simply implementing the program. In addition, a renewed effort may be required each time when some changes, which are likely in a program development cycle, are made to the program. Therefore, the programmer can often feel hesitant to adopt (interactive) theorem proving for detecting program errors in general programming.

We are primarily interested in finding a middle ground. In particular, we are interested in forming a mechanism in a programming language that allows the programmer to provide *key* information needed for establishing program termination

*Partially supported by NSF grant no. CCR-0092703

0-7695-1281-X/01 $10.00 © 2001 IEEE

```
fun ack m n =
  if m = 0 then n+1
  else if n = 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
withtype {i:nat,j:nat} <i,j> => int(i) -> int(j) -> [k:nat] int(k)
```

Figure 1. An implementation of Ackerman function

and then automatically verifies that the provided information indeed suffices. An analogy would be like allowing the user to provide induction hypotheses in inductive theorem proving and then proving theorems with the provided induction hypotheses. Clearly, the challenging question is how such key information for establishing program termination can be formalized and then expressed. The main contribution of this paper lies in our attempt to address the question by presenting a design that allows the programmer to provide through dependent types such key information in a (relatively) simple and clean way.

It is common in practice to prove the termination of recursive functions with metrics. Roughly speaking, we attach a metric in a well-founded ordering to a recursive function and verify that the metric is always decreasing when a recursive function call is made. In this paper, we present an approach that uses the dependent types developed in DML [18, 14] to carry metrics for proving program termination. We form a type system in which metrics can be encoded into types and prove that every well-typed program is terminating. It should be emphasized that we are not here advocating the design of a programming language in which only terminating programs can be written. Instead, we are interested in designing a mechanism in a programming language, which, if the programmer chooses to use it, can facilitate program termination verification. This is to be manifested in that the type system we form can be smoothly embedded into the type system of DML. We now illustrate the basic idea with a concrete example before going into further details.

In Figure 1, an implementation of Ackerman function is given. The `withtype` clause is a type annotation, which states that for natural numbers i and j, this function takes an argument of type $\texttt{int}(i)$ and another argument of type $\texttt{int}(j)$ and returns a natural number as a result. Note that we have refined the usual integer type `int` into infinitely many singleton types $\texttt{int}(a)$ for $a = 0, 1, -1, 2, -2, \ldots$ such that $\texttt{int}(a)$ is precisely the type for integer expressions with value equal to a. We write $\{\texttt{i:nat,j:nat}\}$ for universally quantifying over index variables i and j of sort nat, that is, the sort for index expressions with values being natural numbers. Also, we write $\texttt{[k:nat] int(k)}$ for $\Sigma k : nat.\texttt{int}(k)$, which represents the sum of all types $\texttt{int}(k)$ for $k = 0, 1, 2, \ldots$. The novelty here is the pair $\langle i, j\rangle$ in the type annotation, which indicates that this is the metric to be used for termination checking. We now informally explain how termination checking is performed in this case; assume that i and j are two natural numbers and m and n have types $\texttt{int}(i)$ and $\texttt{int}(j)$, respectively, and attach the metric $\langle i, j\rangle$ to $ack\ m\ n$; note that there are three recursive function calls to ack in the body of ack; we attach the metric $\langle i-1, 1\rangle$ to the first ack since $m-1$ and 1 have types $\texttt{int}(i-1)$ and $\texttt{int}(1)$, respectively; similarly, we attach the metric $\langle i-1, k\rangle$ to the second ack, where k is assumed to be some natural number, and the metric $\langle i, j-1\rangle$ to the third ack; it is obvious that $\langle i-1, 1\rangle < \langle i, j\rangle$, $\langle i-1, k\rangle < \langle i, j\rangle$ and $\langle i, j-1\rangle < \langle i, j\rangle$ hold, where $<$ is the usual lexicographic ordering on pairs of natural numbers; we thus claim that the function ack is terminating (by a theorem proven in this paper). Note that although this is a simple example, its termination cannot be proven with (lexicographical) structural ordering (as the semantic meaning of both addition $+$ and subtraction $-$ is needed).[1]

More realistic examples are to be presented in Section 5, involving dependent datatypes [15], mutual recursion, higher-order functions and polymorphism. The reader may read some of these examples before studying the sections on technical development so as to get a feel as to what can actually be handled by our approach.

Combining metrics with the dependent types in DML poses a number of theoretical and pragmatic questions. We briefly outline our results and design choices.

The first question that arises is to decide what metrics we should support. Clearly, the variety of metrics for establishing program termination is endless in practice. In this paper, we only consider metrics that are tuples of index expressions of sort nat and use the usual lexicographic ordering to compare metrics. The main reasons for this decision are that (a) such metrics are commonly used in practice to establish termination proofs for a large variety of programs and (b) constraints generated from comparing such metrics can be readily handled by the constraint solver *already* built for type-checking DML programs. Note that the usual structural ordering on *first-order* terms can be obtained by attaching to the term the number of constructors in the term, which can be readily accomplished by using the dependent datatype mechanism in DML. However, we are currently unable to capture structural ordering on higher-order terms.

The second question is about establishing the soundness of our approach, that is, proving every well-typed program in the type system we design is terminating. Though the idea mentioned in the example of Ackerman function seems intuitive, this task is far from being trivial because of the presence of higher-order functions. The reader may take a look at the higher-order example in Section 5 to understand this. We seek a method that can be readily adapted to handle various common programming features when they are added,

[1] There is an implementation of Ackerman function that involves only primitive recursion and can thus be easily proven terminating, but the point we drive here is that this particular implementation can be proven terminating with our approach.

including mutual recursion, datatypes, polymorphism, etc. This naturally leads us to the reducibility method [12]. We are to form a notion of reducibility for the dependent types extended with metrics, in which the novelty lies in the treatment of general recursion. This formation, which is novel to our knowledge, constitutes the main technical contribution of the paper.

The third question is about integrating our termination checking mechanism with DML. In practice, it is common to encounter a case where the termination of a function f depends on the termination of another function g, which, unfortunately, is not proven for various reasons, e.g., it is beyond the reach of the adopted mechanism for termination checking or the programmer is simply unwilling to spend the effort proving it. Our approach is designed in a way that allows the programmer to provide a metric in this case for verifying the termination of f conditional on the termination of g, which can still be useful for detecting program errors.

The presented work builds upon our previous work on the use of dependent types in practical programming [18, 14]. While the work has its roots in DML, it is largely unclear, *a priori*, how dependent types in DML can be used for establishing program termination. We thus believe that it is a significant effort to actually design a type system that combines types with metrics and then prove that the type system guarantees program termination. This effort is further strengthened with a prototype implementation and a variety of verified examples.

The rest of the paper is organized as follows. We form a language $\mathrm{ML}_0^{\Pi,\Sigma}$ in Section 2, which essentially extends the simply typed call-by-value λ-calculus with a form of dependent types, developed in DML, and recursion. We then extend $\mathrm{ML}_0^{\Pi,\Sigma}$ to $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ in Section 3, combining metrics with types, and prove that every program in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is terminating. In Section 4, we enrich $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with some significant programming features such as datatypes, mutual recursion and polymorphism. We present some examples in Section 5, illustrating how our approach to program termination verification is applied in practice. We then mention some related work and conclude.

There is a full paper available on-line [16] in which the reader can find details omitted here.

2 $\mathrm{ML}_0^{\Pi,\Sigma}$

We start with a language $\mathrm{ML}_0^{\Pi,\Sigma}$, which essentially extends the simply typed call-by-value λ-calculus with a form of dependent types and (general) recursion. The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$ is given in Figure 2.

2.1 Syntax

We fix an integer domain and restrict type index expressions, namely, the expressions that can be used to index a type, to this domain. This is a sorted domain and subset sorts can be formed. For instance, we use *nat* for the subset sort $\{a : int \mid a \geq 0\}$. We use $\delta(\vec{\imath})$ for a base type indexed with a sequence of index expressions $\vec{\imath}$, which may be empty. For instance, $\mathtt{bool}(0)$ and $\mathtt{bool}(1)$ are types for boolean values *false* and *true*, respectively; for each integer i, $\mathtt{int}(i)$ is the singleton type for integer expressions with value equal to i.

We use $\phi \models P$ for a satisfaction relation, which means P holds under ϕ, that is, the formula $(\phi)P$, defined below, is satisfied in the domain of integers.

$$(\cdot)\Phi = \Phi \qquad (\phi, a : int)\Phi = (\phi)\forall a : int.\Phi$$
$$(\phi, a : \{a : \gamma \mid P\})\Phi = (\phi, a : \gamma)(P \supset \Phi)$$
$$(\phi, P)\Phi = (\phi)(P \supset \Phi)$$

For instance, the satisfaction relation

$$a : nat, a \neq 0 \models a - 1 \geq 0$$

holds since the following formula is true in the integer domain.

$$\forall a : int.a \geq 0 \supset (a \neq 0 \supset a - 1 \geq 0)$$

Note that the decidability of the satisfaction relation depends on the constraint domain. For the integer constraint domain we use here, the satisfaction relation is decidable (as we do not accept nonlinear integer constraints).

We use $\Pi a : \gamma.\tau$ and $\Sigma a : \gamma.\tau$ for the usual dependent function and sum types, respectively. A type of form $\Pi\vec{a} : \vec{\gamma}.\tau$ is essentially equivalent to $\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\tau$, where we use $\vec{a} : \vec{\gamma}$ for $a_1 : \gamma_1, \ldots, a_n : \gamma_n$. [2] We also introduce λ-variables and ρ-variables in $\mathrm{ML}_0^{\Pi,\Sigma}$ and use x and f for them, respectively. A lambda-abstraction can only be formed over a λ-variable while recursion (via fixed point operator) must be formed over a ρ-variable. A λ-variable is a value but a ρ-variable is not.

We use λ for abstracting over index variables, **lam** for abstracting over variables, and **fun** for forming recursive functions. Note that the body after either λ or **fun** must be a value. We use $\langle i \mid e \rangle$ for packing an index i with an expression e to form an expression of a dependent sum type, and **open** for unpacking an expression of a dependent sum type.

2.2 Static Semantics

We write $\phi \vdash \tau : *$ to mean that τ is a legally formed type under ϕ and omit the standard rules for such judgments.

index substitutions	θ_I	::=	$[] \mid \theta_I[a \mapsto i]$
substitutions	θ	::=	$[] \mid \theta[x \mapsto e] \mid \theta[f \mapsto e]$

A substitution is a finite mapping and [] represents an empty mapping. We use θ_I for a substitution mapping index variables to index expressions and $\mathbf{dom}(\theta_I)$ for the domain of θ_I. Similar notations are used for substitutions on variables. We write $\bullet[\theta_I]$ ($\bullet[\theta]$) for the result from applying θ_I (θ) to $\bullet$, where $\bullet$ can be a type, an expression, etc. The standard

[2] In practice, we also have types of form $\Sigma\vec{a} : \vec{\gamma}.\tau$, which we omit here for simplifying the presentation.

index constants	c_I	::=	$\cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$
index expressions	i	::=	$a \mid c_I \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1/i_2$
index propositions	P	::=	$i_1 < i_2 \mid i_1 \leq i_2 \mid i_1 > i_2 \mid i_1 \geq i_2 \mid i_1 = i_2 \mid i_1 \neq i_2 \mid P_1 \wedge P_2 \mid P_1 \vee P_2$
index sorts	γ	::=	$int \mid \{a : \gamma \mid P\}$
index variable contexts	ϕ	::=	$\cdot \mid \phi, a : \gamma \mid \phi, P$
index constraints	Φ	::=	$P \mid P \supset \Phi \mid \forall a : \gamma.\Phi$
types	τ	::=	$\delta(\vec{\imath}) \mid \Pi\vec{a} : \vec{\gamma}.\tau \mid \Sigma a : \gamma.\tau$
contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau \mid \Gamma, f : \tau$
constants	c	::=	$true \mid false \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \cdots$
expressions	e	::=	$c \mid x \mid f \mid \mathbf{if}(e, e_1, e_2) \mid \lambda\vec{a} : \vec{\gamma}.v \mid \mathbf{lam}\ x : \tau.e \mid e_1(e_2) \mid$ $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v \mid e[\vec{\imath}] \mid \langle i \mid e \rangle \mid \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2$
values	v	::=	$c \mid x \mid \lambda\vec{a} : \vec{\gamma}.v \mid \mathbf{lam}\ x : \tau.e \mid \langle i \mid v \rangle$

Figure 2. The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \vdash \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2}\ \textbf{(type-eq)} \qquad \frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau}\ \textbf{(type-}\lambda\textbf{-var)} \qquad \frac{\Gamma(f) = \tau}{\phi;\Gamma \vdash f : \tau}\ \textbf{(type-}\rho\textbf{-var)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma};\Gamma \vdash v : \tau}{\phi;\Gamma \vdash \lambda\vec{a} : \vec{\gamma}.v : \Pi\vec{a} : \vec{\gamma}.\tau}\ \textbf{(type-ilam)} \qquad \frac{\phi;\Gamma \vdash e : \Pi\vec{a} : \vec{\gamma}.\tau \quad \phi \vdash \vec{\imath} : \vec{\gamma}}{\phi;\Gamma \vdash e[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}]}\ \textbf{(type-iapp)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma};\Gamma, f : \Pi\vec{a} : \vec{\gamma}.\tau \vdash v : \tau}{\phi;\Gamma \vdash \mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v : \Pi\vec{a} : \vec{\gamma}.\tau}\ \textbf{(type-fun)}$$

$$\frac{\phi;\Gamma \vdash e : \mathtt{bool}(i) \quad \phi, i = 1;\Gamma \vdash e_1 : \tau \quad \phi, i = 0;\Gamma \vdash e_2 : \tau}{\phi;\Gamma \vdash \mathbf{if}(e, e_1, e_2) : \tau}\ \textbf{(type-if)}$$

$$\frac{\phi;\Gamma, x : \tau_1 \vdash e : \tau_2}{\phi;\Gamma \vdash \mathbf{lam}\ x : \tau_1.e : \tau_1 \to \tau_2}\ \textbf{(type-lam)} \qquad \frac{\phi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \phi;\Gamma \vdash e_2 : \tau_1}{\phi;\Gamma \vdash e_1(e_2) : \tau_2}\ \textbf{(type-app)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \quad \phi, a : \gamma;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2 : \tau_2}\ \textbf{(type-open)} \qquad \frac{\phi \vdash i : \gamma \quad \phi;\Gamma \vdash e : \tau[a \mapsto i]}{\phi;\Gamma \vdash \langle i \mid e \rangle : \Sigma a : \gamma.\tau}\ \textbf{(type-pack)}$$

Figure 3. Typing Rules for $\mathrm{ML}_0^{\Pi,\Sigma}$

definition is omitted. The following rules are for judgments of form $\phi \vdash \theta_I : \phi'$, which roughly means that θ_I has "type" ϕ'.

$$\frac{}{\phi \vdash [] : \cdot}\ \textbf{(sub-i-empty)}$$

$$\frac{\phi \vdash \theta_I : \phi' \quad \phi \vdash i : \gamma[\theta_I]}{\phi \vdash \theta_I[a \mapsto i] : \phi', a : \gamma}\ \textbf{(sub-i-var)}$$

$$\frac{\phi \vdash \theta_I : \phi' \quad \phi \models P[\theta_I]}{\phi \vdash \theta_I : \phi', P}\ \textbf{(sub-i-prop)}$$

We write $\mathbf{dom}(\Gamma)$ for the domain of Γ, that is, the set of variables declared in Γ. Given substitutions θ_I and θ, we say $\phi;\Gamma \vdash (\theta_I;\theta) : (\phi';\Gamma')$ holds if $\phi \vdash \theta_I : \phi'$ and $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ and $\phi;\Gamma \vdash \theta(x) : \Gamma'(x)[\theta_I]$ for all $x \in \mathbf{dom}(\Gamma')$.

We write $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i = j$ from index expressions to types, determined by the following rules. It is the application of these rules that generates constraints during type-checking.

$$\frac{\phi \models i = j}{\phi \models \delta(i) \equiv \delta(j)} \qquad \frac{\phi \models \tau_1' \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}$$

$$\frac{\phi, \vec{a} : \vec{\gamma} \models \tau \equiv \tau'}{\phi \models \Pi\vec{a} : \vec{\gamma}.\tau \equiv \Pi\vec{a} : \vec{\gamma}.\tau'} \qquad \frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Sigma a : \gamma.\tau \equiv \Sigma a : \gamma.\tau'}$$

We present the typing rules for $\mathrm{ML}_0^{\Pi,\Sigma}$ in Figure 3. Some of these rules have obvious side conditions, which are omitted. For instance, in the rule **(type-ilam)**, $\vec{a}$ cannot have free occurrences in Γ. The following lemma plays a pivotal rôle in proving the subject reduction theorem for $\mathrm{ML}_0^{\Pi,\Sigma}$, whose standard proof is available in [14].

Lemma 2.1 *Assume $\phi, \phi';\Gamma, \Gamma' \vdash e : \tau$ is derivable and $\phi;\Gamma \vdash (\theta_I;\theta) : (\phi';\Gamma')$ holds. Then we can derive $\phi;\Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I]$.*

2.3 Dynamic Semantics

We present the dynamic semantics of $\mathrm{ML}_0^{\Pi,\Sigma}$ through the use of evaluation contexts defined below. Certainly, there are other possibilities for this purpose, which we do not explore here. [3]

$$\begin{array}{l}\text{evaluation contexts } E ::= \\ \quad [] \mid \mathbf{if}(E, e_1, e_2) \mid E[\vec{\imath}] \mid E(e) \mid v(E) \mid \\ \quad \langle i \mid E \rangle \mid \mathbf{open}\ E\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e\end{array}$$

We write $E[e]$ for the expression resulting from replacing the hole [] in E with e. Note that this replacement can *never* result in capturing free variables.

Definition 2.2 *A redex is defined below.*

- $\mathbf{if}(c, e_1, e_2)$ *are redexes for* $c = true, false$, *which reduce to* e_1 *and* e_2, *respectively.*
- $(\mathbf{lam}\ x : \tau.e)(v)$ *is a redex, which reduces to* $e[x \mapsto v]$.
- *Let* e *be* $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v$ *Then* e *is a redex, which reduces to* $\lambda\vec{a} : \vec{\gamma}.v[f \mapsto e]$.
- $(\lambda\vec{a} : \vec{\gamma}.v)[\vec{\imath}]$ *is a redex, which reduces to* $v[\vec{a} \mapsto \vec{\imath}]$.
- $\mathbf{open}\ \langle i \mid v \rangle\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e$ *is a redex, which reduces to* $e[a \mapsto i][x \mapsto v]$.

We use r *for a redex and write* $r \hookrightarrow e$ *if* r *reduces to* e. *If* $e_1 = E[r]$, $e_2 = E[e]$ *and* $r \hookrightarrow e$, *we write* $e_1 \hookrightarrow e_2$ *and say* e_1 *reduces to* e_2 *in one step.*

Let $\hookrightarrow^*$ be the reflexive and transitive closure of $\hookrightarrow$. We say e_1 reduces to e_2 (in many steps) if $e_1 \hookrightarrow^* e_2$. We omit the standard proof for the following subject reduction theorem, which uses Lemma 2.1.

Theorem 2.3 *(Subject Reduction) Assume* $\cdot;\cdot \vdash e : \tau$ *is derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$. *If* $e \hookrightarrow^* e'$, *then* $\cdot;\cdot \vdash e' : \tau$ *is also derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$.

2.4 Erasure

We can simply transform $\mathrm{ML}_0^{\Pi,\Sigma}$ into a language ML_0 by erasing all syntax related to type index expressions in $\mathrm{ML}_0^{\Pi,\Sigma}$. Then ML_0 basically extends simply typed λ-calculus with recursion. Let $|e|$ be the erasure of expression e. We have e_1 reducing to e_2 in $\mathrm{ML}_0^{\Pi,\Sigma}$ implies $|e_1|$ reducing to $|e_2|$ in ML_0. Therefore, if e is terminating in $\mathrm{ML}_0^{\Pi,\Sigma}$ then $|e|$ is terminating in ML_0. This is a crucial point since the evaluation of a program in $\mathrm{ML}_0^{\Pi,\Sigma}$ is (most likely) done through the evaluation of its erasure in ML_0. Please find more details on this issue in [18, 14].

[3] For instance, it is suggested that one present the dynamic semantics in the style of natural semantics and then later form the notion of reducibility for evaluation rules.

3 $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$

We combine metrics with the dependent types in $\mathrm{ML}_0^{\Pi,\Sigma}$, forming a language $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$. We then prove that every well-typed program in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is terminating, which is the main technical contribution of the paper.

3.1 Metrics

We use $\leq$ for the usual lexicographic ordering on tuples of natural numbers and $<$ for the strict part of $\leq$. Given two tuples of natural numbers $\langle i_1, \ldots, i_n \rangle$ and $\langle i'_1, \ldots, i'_{n'} \rangle$, $\langle i_1, \ldots, i_n \rangle < \langle i'_1, \ldots, i'_{n'} \rangle$ holds if $n = n'$ and for some $0 \leq k \leq n$, $i_j = i'_j$ for $j = 1, \ldots, k-1$ and $i_k < i'_k$. Evidently, $<$ is a well-founded. We stress that (in theory) there is no difficulty supporting various other well-founded orderings on natural numbers such as the usual multiset ordering. We fix an ordering solely for easing the presentation.

Definition 3.1 *(Metric) Let* $\mu = \langle i_1, \ldots, i_n \rangle$ *be a tuple of index expressions and* ϕ *be an index variable context. We say* μ *is a metric under* ϕ *if* $\phi \vdash i_j : nat$ *are derivable for* $j = 1, \ldots, n$. *We write* $\phi \vdash \mu : metric$ *to mean* μ *is a metric under* ϕ.

A decorated type in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is of form $\Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$, and the following rule is for forming such types.

$$\frac{\phi, \vec{a} : \vec{\gamma} \vdash \tau : * \quad \phi, \vec{a} : \vec{\gamma} \vdash \mu : metric}{\phi \vdash \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau : *}$$

The syntax of $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is the same as that of $\mathrm{ML}_0^{\Pi,\Sigma}$ except that a context Γ in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ maps every ρ-variable f in its domain to a decorated type and a recursive function in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is of form $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \mathbf{is}\ v$. The process of translating a source program into an expression in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is what we call *elaboration*, which is thoroughly explained in [18, 14]. Our approach to program termination verification is to be applied to elaborated programs.

3.2 Dynamic and Static Semantics

The dynamic semantics of $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is formed in precisely the same manner as that of $\mathrm{ML}_0^{\Pi,\Sigma}$ and we thus omit all the details.

The difference between $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ and $\mathrm{ML}_0^{\Pi,\Sigma}$ lies in static semantics. There are two kinds of typing judgments in $\mathrm{ML}_0^{\Pi,\Sigma}$, which are of forms $\phi;\Gamma \vdash e : \tau$ and $\phi;\Gamma \vdash e : \tau \ll_f \mu_0$. We call the latter a metric typing judgment, for which we give some explanation. Suppose $\phi;\Gamma \vdash e : \tau \ll_f \mu_0$ and $\Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$; roughly speaking, for each free occurrence of f in e, f is followed by a sequence of index expressions $[\vec{\imath}]$ such that $\mu[\vec{a} \mapsto \vec{\imath}]$, which we call the label of this occurrence of f, is less than μ_0 under ϕ. Now suppose we have a well-typed closed recursive function

$e = \mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \mathbf{is}\ v$ in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ and $\vec{\imath}$ are of sorts $\vec{\gamma}$; then $f[\vec{\imath}][f \mapsto e] = e[\vec{\imath}] \hookrightarrow^* v[\vec{a} \mapsto \vec{\imath}][f \mapsto e]$ holds; by the rule **(type-fun)**, we know that all labels of f in v are less than $\mu[\vec{a} \mapsto \vec{\imath}]$, which is the label of f in $f[\vec{\imath}]$; since labels cannot decrease forever, this yields some basic intuition on why all recursive functions in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ are terminating. However, this intuitive argument is difficult to be formalized directly in the presence of high-order functions.

The typing rules in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ for a judgment of form $\phi; \Gamma \vdash e : \tau$ are essentially the same as those in $\mathrm{ML}_0^{\Pi.\Sigma}$ except the following ones.

$$\frac{\Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau}{\phi; \Gamma \vdash f : \Pi\vec{a} : \vec{\gamma}.\tau}\ \textbf{(type-}\rho\textbf{-var)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma, f : \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau \vdash v : \tau \ll_f \mu}{\phi; \Gamma \vdash \mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \mathbf{is}\ v : \Pi\vec{a} : \vec{\gamma}.\tau}\ \textbf{(type-fun)}$$

We present the rules for deriving metric typing judgments in Figure 4. Given $\mu = \langle i_1, \ldots, i_n \rangle$ and $\mu' = \langle i'_1, \ldots, i'_n \rangle$, $\phi \models \mu < \mu'$ means that for some $1 \leq k < n$, $\phi, i_1 = i'_1, \ldots, i_{j-1} = i'_{j-1} \models i_j \leq i'_j$ are satisfied for all $1 \leq j < k$ and $\phi, i_1 = i'_1, \ldots, i_{k-1} = i'_{k-1} \models i_k < i'_k$ is also satisfied.

Lemma 3.2 *We have the following.*

1. *Assume $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds. Then we can derive $\phi; \Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I]$.*
2. *Assume $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau \ll_f \mu$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds and $f \in \mathbf{dom}(\Gamma)$. Then we can derive $\phi; \Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I] \ll_f \mu[\theta_I]$.*

Proof (1) and (2) are proven simultaneously by structural induction on derivations of $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ and $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau \ll_f \mu$, respectively. ∎

Theorem 3.3 *(Subject Reduction) Assume $\cdot; \cdot \vdash e : \tau$ is derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$. If $e \hookrightarrow^* e'$, then $\cdot; \cdot \vdash e' : \tau$ is also derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.*

Obviously, we have the following.

Proposition 3.4 *Assume that $\mathcal{D}$ is a derivation $\phi; \Gamma \vdash e : \tau \ll_f \mu_0$. Then then there is a derivation of $\phi; \Gamma \vdash e : \tau$ with the same height[4] as $\mathcal{D}$.*

3.3 Reducibility

We define the notion of reducibility for well-typed closed expressions.

Definition 3.5 *(Reducibility) Suppose that e is a closed expression of type τ and $e \hookrightarrow^* v$ holds for some value v. The reducibility of e is defined by induction on the complexity of τ.*

1. *τ is a base type. Then e is reducible.*
2. *$\tau = \tau_1 \to \tau_2$. Then e is reducible if $e(v_1)$ are reducible for all reducible values v_1 of type τ.*
3. *$\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$. Then e is reducible if $e[\vec{\imath}]$ are reducible for all $\vec{\imath} : \vec{\gamma}$.*
4. *$\tau = \Sigma a : \gamma.\tau_1$. Then e is reducible if $v = \langle i \mid v_1 \rangle$ for some i and v_1 such that v_1 is a reducible value of type $\tau_1[a \mapsto i]$.*

Note that reducibility is *only* defined for closed expressions that reduce to values.

Proposition 3.6 *Assume that e is a closed expression of type τ and $e \hookrightarrow e'$ holds. Then e is reducible if and only if e' is reducible.*

Proof By induction on the complexity of τ. ∎

The following is a key notion for handling recursion, which, though natural, requires some technical insights.

Definition 3.7 *(μ-Reducibility). Let e be a well-typed closed recursive function $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \mathbf{is}\ v$ and μ_0 be a closed metric. e is μ_0-reducible if $e[\vec{\imath}]$ are reducible for all $\vec{\imath} : \vec{\gamma}$ satisfying $\mu[\vec{a} \mapsto \vec{\imath}] < \mu_0$.*

Definition 3.8 *Let θ be a substitution that maps variables to expressions; for every $x \in \mathbf{dom}(\theta)$, θ is x-reducible if $\theta(x)$ is reducible; for every $f \in \mathbf{dom}(\theta)$, θ is (f, μ_f)-reducible if $\theta(f)$ is μ_f-reducible.*

In some sense, the following lemma verifies whether the notion of reducibility is formed correctly, where the difficulty probably lies in its formulation rather than in its proof.

Lemma 3.9 *(Main Lemma) Assume that $\phi; \Gamma \vdash e : \tau$ and $\cdot \vdash (\theta_I; \theta) : (\phi; \Gamma)$ are derivable. Also assume that θ is x-reducible for every $x \in \mathbf{dom}(\Gamma)$ and for every $f \in \mathbf{dom}(\Gamma)$, $\cdot; \Gamma[\theta_I] \vdash e[\theta_I] : \tau[\theta_I] \ll_f \mu_f$ is derivable and θ is (f, μ_f)-reducible. Then $e[\theta_I][\theta]$ is reducible.*

Proof Let $\mathcal{D}$ be a derivation of $\phi; \Gamma \vdash e : \tau$ and we proceed by induction on the height of $\mathcal{D}$. We present the most interesting case below. All other cases can be found in [16]. Assume that the following rule **(type-fun)** is last applied in $\mathcal{D}$,

$$\frac{\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\mu_1 \Rightarrow \tau_1 \vdash v_1 : \tau_1 \ll_{f_1} \mu_1}{\phi; \Gamma \vdash \mathbf{fun}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1\ \mathbf{is}\ v_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1}$$

where we have $e = \mathbf{fun}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1\ \mathbf{is}\ v_1$ and $\tau = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1$. Suppose that $e^* = e[\theta_I][\theta]$ is not reducible. Then by definition there exist $\vec{\imath}_0 : \vec{\gamma}_1^*$ such that $e^*[\vec{\imath}_0]$ is not reducible but $e^*[\vec{\imath}]$ are reducible for all $\vec{\imath} : \vec{\gamma}_1^*$ satisfying $\mu_1^*[\vec{a}_1 \mapsto \vec{\imath}] < \mu_1^*[\vec{a}_1 \mapsto \vec{\imath}_0]$, where $\vec{\gamma}_1^* = \vec{\gamma}_1[\theta_I]$ and $\mu_1^* = \mu_1[\theta_I]$. In other words, e^* is μ_{f_1}-reducible for $\mu_{f_1} = \mu_1^*[\vec{a}_1 \mapsto \vec{\imath}_0]$. Note that we can derive $\cdot; \Gamma[\theta_I], f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1^*.\tau_1[\theta_I] \vdash v_1[\theta_I[\vec{a}_1 \mapsto \vec{\imath}_0]] : \tau_1[\theta_I[\vec{a}_1 \mapsto$

[4]For a minor technicality reason, we count neither of the rules **(type-ρ-var)** and **($\ll$-ρ-var)** when calculating the height of a derivation.

$$\frac{\Gamma(x) = \tau}{\phi; \Gamma \vdash x : \tau \ll_f \mu_0} \text{ (}\ll\text{-}\lambda\text{-var)} \qquad \frac{\Gamma(f_1) = \tau \quad f_1 \neq f}{\phi; \Gamma \vdash f_1 : \tau \ll_f \mu_0} \text{ (}\ll\text{-}\rho\text{-var)}$$

$$\frac{\phi; \Gamma \vdash e : \mathtt{bool}(i) \ll_f \mu_0 \quad \phi, i = 1; \Gamma \vdash e_1 : \tau \ll_f \mu_0 \quad \phi, i = 0; \Gamma \vdash e_2 : \tau \ll_f \mu_0}{\phi; \Gamma \vdash \mathbf{if}(e, e_1, e_2) : \tau \ll_f \mu_0} \text{ (}\ll\text{-if)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma \vdash v : \tau \ll_f \mu_0}{\phi; \Gamma \vdash \lambda \vec{a} : \vec{\gamma}.v : \Pi \vec{a} : \vec{\gamma}.\tau \ll_f \mu_0} \text{ (}\ll\text{-ilam)} \qquad \frac{\phi; \Gamma \vdash e : \Pi \vec{a} : \vec{\gamma}.\tau \ll_f \mu_0 \quad \phi \vdash \vec{\imath} : \vec{\gamma}}{\phi; \Gamma \vdash e[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0} \text{ (}\ll\text{-iapp)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e : \tau_2 \ll_f \mu_0}{\phi; \Gamma \vdash \mathbf{lam}\ x : \tau_1.e : \tau_1 \to \tau_2 \ll_f \mu_0} \text{ (}\ll\text{-lam)} \qquad \frac{\phi; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \ll_f \mu_0 \quad \phi; \Gamma \vdash e_2 : \tau_1 \ll_f \mu_0}{\phi; \Gamma \vdash e_1(e_2) : \tau_2 \ll_f \mu_0} \text{ (}\ll\text{-app)}$$

$$\frac{\begin{array}{c}\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi \vec{a}_1 : \mu_1 \Rightarrow \vec{\gamma}_1.\tau_1 \vdash v_1 : \tau_1 \ll_{f_1} \mu_1 \\ \phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi \vec{a}_1 : \vec{\gamma}_1.\tau_1 \vdash e_1 : \tau_1 \ll_f \mu_0\end{array}}{\phi; \Gamma \vdash \mathbf{fun}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1\ \mathbf{is}\ v_1 : \Pi \vec{a}_1 : \vec{\gamma}_1.\tau_1 \ll_f \mu_0} \text{ (}\ll\text{-fun)}$$

$$\frac{\phi \vdash \vec{\imath} : \vec{\gamma} \quad \phi \models \mu[\vec{a} \mapsto \vec{\imath}] < \mu_0 \quad \Gamma(f) = \Pi \vec{a} : \vec{\gamma}.\mu \Rightarrow \tau}{\phi; \Gamma \vdash f[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0} \text{ (}\ll\text{-lab)}$$

$$\frac{\phi \vdash i : \gamma \quad \phi; \Gamma \vdash e : \tau[a \mapsto i] \ll_f \mu_0}{\phi; \Gamma \vdash \langle i \mid e \rangle : \Sigma a : \gamma.\tau \ll_f \mu_0} \text{ (}\ll\text{-pack)}$$

$$\frac{\phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \ll_f \mu_0 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \ll_f \mu_0}{\phi; \Gamma \vdash \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2 : \tau_2 \ll_f \mu_0} \text{ (}\ll\text{-open)}$$

Figure 4. Metric Typing Rules for $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$

$\vec{\imath}_0]] \ll_f \mu_{f_1}$. By Proposition 3.4, there is a derivation $\mathcal{D}_1$ of $\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi \vec{a}_1 : \vec{\gamma}_1.\mu_1 \Rightarrow \tau_1 \vdash v_1 : \tau_1$ such that the height of $\mathcal{D}_1$ is less than that of $\mathcal{D}$. By induction hypothesis, we have that $v_1^* = v_1[\theta_I[\vec{a}_1 \mapsto \vec{\imath}_0]][\theta[f_1 \mapsto e^*]]$ is reducible. Note that $e^*[\vec{\imath}_0] \hookrightarrow^* v_1^*$ and thus $e^*[\vec{\imath}_0]$ is reducible, contradicting the definition of $\vec{\imath}_0$. Therefore, e^* is reducible. ∎

The following is the main result of the paper.

Corollary 3.10 *If $\cdot; \cdot \vdash e : \tau$ is derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$, then e in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is reducible and thus reduces to a value.*

Proof The corollary follows from Lemma 3.9. ∎

4 Extensions

We can extend $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with some significant programming features such as mutual recursion, datatypes and polymorphism, defining the notion of reducibility for each extension and thus making it clear that Lemma 3.9 still holds after the extension. We present in this section the treatment of mutual recursion and currying, leaving the details in [16].

4.1 Mutual Recursion

The treatment of mutual recursion is slightly different from the standard one. The syntax and typing rules for handling mutual recursion are given in Figure 5. We use $(\tau_1, \ldots, \tau_n)$ for the type of an expression representing n mutually recursive functions of types $\tau_1, \ldots, \tau_n$, respectively, which should not be confused with the product of types $\tau_1, \ldots, \tau_n$. Also, the n in $e.n$ must be a positive (constant) integer. Let v be the following expression.

$$\mathbf{funs}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \tau_1\ \mathbf{is}\ v_1\ \mathbf{and} \ldots \mathbf{and}\ f_n[\vec{a}_n : \vec{\gamma}_n] : \tau_n\ \mathbf{is}\ v_n$$

Then for every $1 \leq k \leq n$, $v.k$ is a redex, which reduces to $\lambda \vec{a}_k : \vec{\gamma}_k.v_k[f_1 \mapsto v.1, \ldots, f_n \mapsto v.n]$. Let $\vec{f} = f_1, \ldots, f_n$ and we form a metric typing judgment $\phi; \Gamma \vdash e \ll_{\vec{f}} \mu_0$ for verifying that all labels of $f_1, \ldots, f_n$ in e are less than μ_0 under ϕ. The rules for deriving such a judgment are essentially the same as those in Figure 4 except (**≪-lab**), which is given below.

$$\frac{f \text{ in } \vec{f} \quad \Gamma(f) = \Pi \vec{a} : \vec{\gamma}.\mu \Rightarrow \tau \quad \phi \models \mu[\vec{a} \mapsto \vec{\imath}] < \mu_0}{\phi; \Gamma \vdash f[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}] \ll_{\vec{f}} \mu_0}$$

The rule (**≪-funs**) for handling mutual recursion is straightforward and thus omitted.

Definition 4.1 *(Reducibility) Let e be a closed expression of type $(\tau_1, \ldots, \tau_n)$ and e reduces to v. e is reducible if e.k are reducible for $k = 1, \ldots, n$.*

4.2 Currying

A decorated type must so far be of form $\Pi \vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$ and this restriction has a rather unpleasant consequence. For

$$
\begin{array}{lrcl}
\text{types} & \tau & ::= & \cdots \mid (\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1, \ldots, \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n) \\
\text{expressions} & e & ::= & \cdots \mid e.n \mid \textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \tau_n \textbf{ is } v_n \\
\text{values} & v & ::= & \cdots \mid \textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \tau_n \textbf{ is } v_n
\end{array}
$$

$$
\frac{\begin{array}{c}\vec{f} = f_1, \ldots, f_n \quad \tau = (\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1, \ldots, \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n) \\ \phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1 : \mu_1 \Rightarrow \tau_1, \ldots, f_n : \Pi\vec{a}_n : \vec{\gamma}_n : \mu_n \Rightarrow \tau_n \vdash v_1 : \tau_1 \ll_{\vec{f}} \mu_1 \\ \cdots \quad \cdots \\ \phi, \vec{a}_n : \vec{\gamma}_n; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1 : \mu_1 \Rightarrow \tau_1, \ldots, f_n : \Pi\vec{a}_n : \vec{\gamma}_n : \mu_n \Rightarrow \tau_n \vdash v_n : \tau_n \ll_{\vec{f}} \mu_n\end{array}}{\phi; \Gamma \vdash \textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \mu_n \Rightarrow \tau_n \textbf{ is } v_n : \tau} \textbf{ (type-funs)}
$$

$$
\frac{\phi; \Gamma \vdash e : (\tau_1, \ldots, \tau_n) \quad 1 \leq k \leq n}{\phi; \Gamma \vdash e.k : \tau_k} \textbf{ (type-choose)}
$$

Figure 5. The Syntax and Typing Rules for Mutual Recursion

instance, we may want to assign the following type τ to the implementation of Ackerman function in Figure 1:

```
{i:nat} int(i) -> {j:nat} int(j) -> int,
```

which is formally written as

$$\Pi a_1 : nat.\mathtt{int}(a_1) \to \Pi a_2 : nat.\mathtt{int}(a_2) \to \Sigma a : nat.\mathtt{int}(a).$$

If we decorate τ with a metric μ, then μ can only involve the index variable a_1, making it impossible to verify that the implementation is terminating.

We generalize the form of decorated types to the following so as to address the problem.

$$\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \to \cdots \to \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n \to \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau.$$

Also, we introduce the following form of expression e for representing a recursive function.

$$\textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1) \cdots [\vec{a}_n : \vec{\gamma}_n](x_n : \tau_n)[\vec{a} : \vec{\gamma}] : \tau \textbf{ is } e_0$$

We require that e_0 be a value if $n = 0$. In the following, we only deal with the case $n = 1$. For $n > 1$, the treatment is similar. For $e = \textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \tau \textbf{ is } e_0$, we have $e \hookrightarrow \lambda\vec{a}_1 : \vec{\gamma}_1.\textbf{lam } x_1 : \tau_1.\lambda\vec{a} : \vec{\gamma}.e_0$ and the following typing rule

$$\frac{\phi, \vec{a}_1 : \vec{\gamma}_1, \vec{a} : \vec{\gamma}; \Gamma, f : \tau_0, x_1 : \tau_1 \vdash e : \tau \ll_f \mu}{\phi; \Gamma \vdash \textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau \textbf{ is } e : \tau_0}$$

where $\tau_0 = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \to \Pi\vec{a} : \vec{\gamma}.\tau$, and the following metric typing rule

$$\frac{\begin{array}{c}\phi \models \vec{\imath}_1 : \vec{\gamma}_1 \quad \phi \models \vec{\imath} : \vec{\gamma}[\vec{a}_1 \mapsto \vec{\imath}_1] \\ \phi \models \mu[\vec{\imath}_1 \mapsto \vec{a}_1][\vec{a} \mapsto \vec{\imath}] < \mu_0 \\ \phi; \Gamma \vdash e_1 : \tau_1[\vec{a}_1 \mapsto \vec{\imath}_1] \ll_f \mu_0 \\ \Gamma(f) = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \to \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau\end{array}}{\phi; \Gamma \vdash f[\vec{\imath}_1](e_1)[\vec{\imath}] : \tau[\vec{a}_1 \mapsto \vec{\imath}_1][\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0}$$

Definition 4.2 *(μ-reducibility) Let e be a closed recursive function* $\textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \tau \textbf{ is } e$ *and μ_0 be a closed metric. e is μ_0-reducible if $e[\vec{\imath}_1](v)[\vec{\imath}]$ are reducible for all reducible values $v : \tau_1[\vec{a}_1 \mapsto \vec{\imath}_1]$ and $\vec{\imath}_1 : \vec{\gamma}_1$ and $\vec{\imath} : \vec{\gamma}[\vec{a}_1 \mapsto \vec{\imath}_1]$ satisfying $\mu[\vec{a}_1 \mapsto \vec{\imath}_1][\vec{a} \mapsto \vec{\imath}] < \mu_0$.*

5 Practice

We have implemented a type-checker for $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ in a prototype implementation of DML and experimented with various examples, some of which are presented below. We also address the practicality issue at the end of this section.

5.1 Examples

We demonstrate how various programming features are handled in practice by our approach to program termination verification.

Primitive Recursion The following is an implementation of the primitive recursion operator R in Gödel's $\mathcal{T}$, which is clearly typable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$. Note that Z and S are assigned the types $Nat(0)$ and $\Pi n : nat.Nat(n) \to Nat(n+1)$, respectively.

```
datatype Nat with nat =
  Z(0) | {n:nat} S(n+1) of Nat(n)

fun('a) R Z u v =
  u | R (S n) u v = v n (R n u v)
withtype
 {n:nat} <n> =>
 Nat(n) -> 'a -> (Nat -> 'a -> 'a) -> 'a
(* Nat is for [n:nat] Nat(n) in a type *)
```

By Corollary 3.10, it is clear that every term in $\mathcal{T}$ is terminating (or weakly normalizing). This is the only example in this paper that can be proven terminating with a structural ordering. The point we make is that though it seems "evident" that the use of R cannot cause non-termination, it is not trivial at all to prove every term in $\mathcal{T}$ is terminating. Notice that such a proof cannot be obtained in Peano arithmetic. The notion of reducibility is precisely invented for overcoming the difficulty [12]. Actually, every term in $\mathcal{T}$ is strongly normalizing, but this obviously is untrue in

$\mathrm{ML}^{\Pi,\Sigma}_{0,\ll}$.

Nested Recursive Function Call The program in Figure 6 involving a nested recursive function call implements McCarthy's "91" function. The `withtype` clause indicates that for every integer x, $f91(x)$ returns integer 91 if $x \leq 100$ and $x - 10$ if $x \geq 101$. We informally explain why the metric in the type annotation suffices to establish the termination of $f91$; for the inner call to $f91$, we need to prove that $\phi \models \max(0, 101 - (i + 11)) < \max(0, 101 - i)$ is satisfied for $\phi = i : int, i \leq 100$, which is obvious; for the outer call to $f91$, we need to verify that $\phi_1 \models \max(0, 101 - j) < \max(0, 101 - i)$, where ϕ_1 is $\phi, j : int, P$ and P is

$$(i + 11 \leq 100 \wedge j = 91) \vee (i + 11 \geq 101 \wedge j = i + 11 - 10)$$

If $i + 11 \leq 100$, then $j = 91$ and $\max(0, 101 - j) = 10 < 12 \leq 101 - i$; if $i + 11 \geq 101$, then $j = i + 11 - 10 = i + 1$ and $\max(0, 101 - j) < 101 - i$ (since $i \leq 100$ is assumed in ϕ). Clearly, this example can not be handled with a structural ordering.

Mutual Recursion The program in Figure 7 implements quicksort on a list, where the functions qs and par are defined mutually recursively. We informally explain why this program is typable in $\mathrm{ML}^{\Pi,\Sigma}_{0,\ll}$ and thus qs is a terminating function by Corollary 3.10.

For the call to par in the body of qs, the label is $(0 + 0 + a, a + 1)$, where a is the length of xs'. So we need to verify that $\phi \models (0 + 0 + a, a + 1) < (n, 0)$ is satisfied for $\phi = n : nat, a : nat, a + 1 = n$, which is obvious.

For the two calls to qs in the body of par, we need to verify that $\phi \models (p, 0) < (p + q + r, r + 1)$ and $\phi \models (q, 0) < (p+q+r, r+1)$ for $\phi = p : nat, q : nat, r : nat, r = 0$, both of which hold since $\phi \models p \leq p + q$ and $\phi \models q \leq p + q$ and $\phi \models 0 < 1$. This also indicates why we need $r + 1$ instead of r in the metric for par.

For the two calls to par in the body of par, we need to verify that $\phi \models ((p + 1) + q + a, a) < (p + q + r, r)$ and $\phi \models (p + (q + 1) + a, a) < (p + q + r, r)$ for $\phi = p : nat, q : nat, r : nat, a : nat, r = a + 1$, both of which hold since $\phi \models (p + 1) + q + a = p + q + r$ and $\phi \models p + (q + 1) + a = p + q + r$ and $\phi \models a < r$. Clearly, this example can not be handled with a structural ordering.

Higher-order Function The program in Figure 8 implements a function *accept* that takes a pattern p and a string s and checks whether s matches p, where the meaning of a pattern is given in the comments.

The auxiliary function *acc* is implemented in continuation passing style, which takes a pattern p, a list of characters cs and a continuation k and matches a prefix of cs against p and call k on the rest of characters. Note that k is given a type that allows k to be applied only to a character list not longer than cs. The metric used for proving the termination of *acc* is $\langle n, i\rangle$, where n is the size of p, that is the number constructors in p (excluding *Empty*) and i is the length of cs. Notice the call *acc* p cs' k in the last pattern matching clause; the label attached to this call is $\langle n, i'\rangle$, where i' is the length of cs'; we have $i' \leq i$ since the continuation has the type $\Pi a' : \gamma.(char)list(a') \rightarrow$ `bool`, where γ is $\{a : nat \mid a \leq i\}$; we have $i \neq i'$ since $length(cs') = length(cs)$ must be false when this call happens; therefore we have $i' < i$ [5] and then $\langle n, i'\rangle < \langle n, i\rangle$. It is straightforward to see that the labels attached to other calls to *acc* are less than $\langle n, i\rangle$. By Corollary 3.10, *acc* is terminating, which implies that *accept* is terminating (assuming *explode* is terminating). In every aspect, this is a non-trivial example even for interactive theorem proving systems.

Notice that the test $length(cs') = length(cs)$ in the body of *acc* can be time-consuming. This can be resolved by using a continuation that accepts as its arguments both a character list and its length. In [5], there is an elegant implementation of *accept* that does some processing on the pattern to be matched and then eliminates the test.

Run-time Check There are also realistic cases where termination depends on a program invariant that cannot (or is difficult to) be captured in the type system of DML. For instance, the following example is adopted from an implementation of bit reversing, which is a part of an implementation of fast Fourier transform (FFT).

```
fun loop (j, k) =
  if (k<j) then loop (j-k, k/2) else j+k
withtype
  {a:nat,b:nat} int(a) * int(b) -> int
```

Obviously, $loop(1, 0)$ is not terminating. However, we may know for some reason that the second argument of *loop* can never be 0 during execution. This leads to the following implementation, in which we need to check that $k > 1$ holds before calling $loop(j - k, k/2)$ so as to guarantee that $k/2$ is a positive integer.

```
fun loop (j, k) =
  if (k < j) then
    if (k > 1) then loop (j - k, k / 2)
    else raise Impossible
  else j + k
withtype {a:nat,b:pos} <max(0, a-b)> =>
         int(a) * int(b) -> int
```

It can now be readily verified that *loop* is a terminating function. This example indicates that we can insert run-time checks to verify program termination, sometimes, approximating a liveness property with a safety property.

5.2 Practicality

There are two separate issues concerning the practicality of our approach to program termination verification, which are (a) the practicality of the termination verification process and (b) the applicability of the approach to realistic programs.

[5] Note that $length(cs')$ and $length(cs)$ have the types `int`(i') and `int`(i), respectively, and thus $length(cs') = length(cs)$ has the type `bool`$(i' = i)$, where $i' = i$ equals 1 or 0 depending on whether i' equals i. Thus, $i' < i$ can be inferred *in the type system*.

```
fun f91 (x) = if (x <= 100) then f91 (f91 (x + 11)) else x - 10
withtype
  {i:int} <max(0, 101-i)> =>
  int(i) -> [j:int | (i<=100 /\ j=91) \/ (i>=101 /\ j=i-10)] int(j)
```

Figure 6. An implementation of McCarthy's "91" function

```
fun('a) qs cmp xs =
  case xs of [] => [] | x :: xs' => par cmp (x, [], [], xs')
withtype ('a * 'a -> bool) -> {n:nat} <n,0> => 'a list(n) -> 'a list(n)

and('a) par cmp (x, l, r, xs) =
  case xs of
    [] => qs cmp l @ (x :: qs cmp r)
  | x' :: xs' => if cmp(x', x) then par cmp (x, x' :: l, r, xs')
                 else par cmp (x, l, x' :: r, xs')
withtype ('a * 'a -> bool) -> {p:nat,q:nat,r:nat} <p+q+r,r+1> =>
         'a * 'a list(p) * 'a list(q) * 'a list(r) -> 'a list(p+q+r+1)
```

Figure 7. An implementation of quicksort on a list

It is easy to observe that the complexity of type-checking in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is basically the same as in $\mathrm{ML}_0^{\Pi,\Sigma}$ since the only added work is to verify that metrics (provided by the programmer) are decreasing, which requires solving some extra constraints. The number of extra constraints generated from type-checking a function is proportional to the number of recursive calls in the body of the function and therefore is likely small. Based on our experience with DML, we thus feel that type-checking in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is suitable for practical use.

As for the applicability of our approach to realistic programs, we use the type system of the programming language C as an example to illustrate a design decision. Obviously, the type system of C is unsound because of (unsafe) type casts, which are often needed in C for typing programs that would otherwise not be possible. In spite of this practice, the type system of C is still of great help for capturing program errors. Clearly, a similar design is to allow the programmer to assert the termination of a function in DML if it cannot be verified, which we may call *termination cast*. Combining termination verification, run-time checks and termination cast, we feel that our approach is promising to be put into practice.

6 Related Work

The amount of research work related to program termination is simply vast. In this section, we mainly mention some related work with which our work shares some similarity either in design or in technique.

Most approaches to automated termination proofs for either programs or term rewriting systems (TRSs) use various heuristics to synthesize well-founded orderings. Such approaches, however, often have difficulty reporting comprehensible information when a program cannot be proven terminating. Following [13], there is also a large amount of work on proving termination of logic programs. In [11], it is reported that the Mercury compiler can perform automated termination checking on realistic logic programs.

However, we address a different question here. We are interested in checking whether a given metric suffices to establish the termination of a program and not in synthesizing such a metric. This design is essentially the same as the one adopted in [10], where it checks whether a given structural ordering (possibly on high-order terms) is decreasing in an inductive proof or a logic program. Clearly, approaches based on checking complements those based on synthesis.

Our approach also relates to the semantic labelling approach [19] designed to prove termination for term rewriting systems (TRSs). The essential idea is to differentiate function calls with labels and show that labels are always decreasing when a function call unfolds. The semantic labelling approach requires constructing a model for a TRS to verify whether labelling is done correctly while our approach does this by type-checking.

The notion of sized types is introduced in [6] for proving the correctness of reactive systems. There, the type system is capable of guaranteeing the termination of well-typed programs. The language presented in [6], which is designed for embedded functional programming, contains a significant restriction as it only supports (a minor variant) of primitive recursion, which can cause inconvenience in programming. For instance, it seems difficult to implement quicksort by using only primitive recursion. From our experience, general recursion is really a major programming feature that greatly complicates program termination verification. Also, the notion of existential dependent types, which we deem indispensable in practical programming, does not exist in [6].

When compared to various (interactive) theorem proving

```
datatype pattern with nat =
  Empty(0) (* empty string matches Empty *)
| Char(1) of char (* "c" matches Char (c) *)
| {i:nat,j:nat} Plus(i+j+1) of pattern(i) * pattern(j)
  (* cs matches Plus(p1, p2) if cs matches either p1 or p2 *)
| {i:nat,j:nat} Times(i+j+1) of pattern(i) * pattern(j)
  (* cs matches Times(p1, p2) if a prefix of cs matches p1 and
     the rest matches p2 *)
| {i:nat} Star(i+1) of pattern(i)
  (* cs matches Star(p) if cs matches some, possibly 0, copies of p *)

(* 'length' computes the length of a list *)
fun('a)
  length (xs) = let
    fun len ([], n) = n
      | len (x :: xs, n) = len (xs, n+1)
    withtype
      {i:nat,j:nat} <i> => 'a list(i) * int(j) -> int(i+j)
   in
      len (xs, 0)
   end
withtype {i:nat} <> => 'a list(i) -> int(i)
(* empty tuple <> is used since 'length' is not recursive *)

fun acc p cs k =
  case p of
    Empty => k (cs)
  | Char(c) =>
    (case cs of
       [] => false
     | c' :: cs' => if (c = c') then k (cs') else false)
  | Plus(p1, p2) => (* in this case, k is used for backtracking *)
    if acc p1 cs k then true else acc p2 cs k
  | Times(p1, p2) => acc p1 cs (fn cs' => acc p2 cs' k)
  | Star(p0) =>
    if k (cs) then true
    else acc p0 cs (fn cs' =>
                      if length(cs') = length(cs) then false
                      else acc p cs' k)
withtype {n:nat} pattern(n) ->
         {i:nat} <n, i> => char list(i) ->
         ({i':nat | i' <= i} char list(i') -> bool) -> bool

(* 'explode' turns a string into a list of characters *)
fun accept p s =
    acc p (explode s) (fn [] => true | _ :: _ => false)
withtype <> => pattern -> string -> bool
```

Figure 8. An implementation of pattern matching on strings

systems such as NuPrl [2], Coq [4], Isabelle [8] and PVS [9], our approach to program termination is weaker (in the sense that [many] fewer programs can be verified terminating) but more automatic and less obtrusive to programming. We have essentially designed a mechanism for program termination verification with a language interface that is to be used during program development cycle. We consider this as the main contribution of the paper. When applied, the designed mechanism intends to facilitate program error detection, leading to the construction of more robust programs.

7 Conclusion and Future Work

We have presented an approach based on dependent types in DML that allows the programmer to supply metrics for verifying program termination and proven its correctness. We have also applied this approach to various examples that involve significant programming features such as a general form of recursion (including mutual recursion), higher-order functions, algebraic datatypes and polymorphism, supporting its usefulness in practice.

A program property is often classified as either a safety property or a liveness property. That a program never performs out-of-bounds array subscripting at run-time is a safety property. It is demonstrated in [17] that dependent types in DML can guarantee that every well-typed program in DML possesses such a safety property, effectively facilitating run-time array bound check elimination. It is, however, unclear (*a priori*) whether dependent types in DML can also be used for establishing liveness properties. In this paper, we have formally addressed the question, demonstrating that dependent types in DML can be combined with metrics to establish program termination, one of the most significant liveness properties.

Termination checking is also useful for compiler optimization. For instance, if one decides to change the execution order of two programs, it may be required to prove that the first program always terminates. Also, it seems feasible to use metrics for estimating the time complexity of programs. In lazy function programming, such information may allow a compiler to decide whether a thunk should be formed. In future, we expect to explore along these lines of research.

Although we have presented many interesting examples that cannot be proven terminating with structural orderings, we emphasize that structural orderings are often effective in practice for establishing program termination. Therefore, it seems fruitful to study a combination of our approach with structural orderings that handles simple cases with either automatically synthesized or manually provided structural orderings and verifies more difficult cases with metrics supplied by the programmer.

References

[1] A. BenCherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *SCP*, 9(2):137–160, 1987.

[2] R. L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[3] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[5] R. Harper. Proof-Directed Debugging. *Journal of Functional Programming*, 9(4):471–477, 1999.

[6] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[7] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science*, pages 402–411, July 1999.

[8] P. Lawrence. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[9] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification, CAV '96*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag LNCS 1102.

[10] B. Pientka and F. Pfenning. Termination and Reduction Checking in the Logical Framework. In *Workshop on Automation of Proofs by Mathematical Induction*, June 2000.

[11] C. Speirs, Z. Somogyi, and H. Søndergarrd. Termination Analysis for Mercury. In *Proceedings of the 4th Static Analysis Symposium*, pages 157–171, September 1997.

[12] W. W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[13] J. D. Ullman and A. V. Gelder. Efficient tests for top-down termination of logic rules. *Journal of the ACM*, 35(2):345–373, 1988.

[14] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[15] H. Xi. Dependently Typed Data Structures. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, pages 17–33, September 1999.

[16] H. Xi. Dependent Types for Program Termination Verification, July 2000. Available as `http://www.ececs.uc.edu/~hwxi/DML/Term`.

[17] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[18] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[19] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

Short Paper Session

Invited Talk

Foundational Proof-Carrying Code

Andrew W. Appel*
Princeton University

Abstract

Proof-carrying code is a framework for the mechanical verification of safety properties of machine language programs, but the problem arises of quis custodiat ipsos custodes*—who will verify the verifier itself? Foundational proof-carrying code is verification from the smallest possible set of axioms, using the simplest possible verifier and the smallest possible runtime system. I will describe many of the mathematical and engineering problems to be solved in the construction of a foundational proof-carrying code system.*

1 Introduction

When you obtain a piece of software – a shrink-wrapped application, a browser plugin, an applet, an OS kernel extension – you might like to ascertain that it's safe to execute: it accesses only its own memory and respects the private variables of the API to which it's linked. In a Java system, for example, the byte-code verifier can make such a guarantee, but only if there's no bug in the verifier itself, or in the just-in-time compiler, or the garbage collector, or other parts of the Java virtual machine (JVM).

If a compiler can produce Typed Assembly Language (TAL) [14], then just by type-checking the low-level representation of the program we can guarantee safety – but only if there's no bug in the typing rules, or in the type-checker, or in the assembler that translates TAL to machine language. Fortunately, these components are significantly smaller and simpler than a Java JIT and JVM.

Proof-carrying code (PCC) [15] constructs and verifies a mathematical proof about the machine-language program itself, and this guarantees safety – but only if there's no bug in the verification-condition generator, or in the logical axioms, or the typing rules, or the proof-checker.

What is the minimum possible size of the components that must be trusted in a PCC system? This is like asking, what is the minimum set of axioms necessary to prove a particular theorem? A foundational proof is one from just the foundations of mathematical logic, without additional axioms and assumptions; foundational proof-carrying code is PCC with trusted components an order of magnitude smaller than previous PCC systems.

Conventional proof-carrying code. Necula [15] showed how to specify and verify safety properties of machine-language programs to ensure that an untrusted program does no harm – does not access unauthorized resources, read private data, or overwrite valuable data. The provider of a PCC program must provide both the executable code and a machine-checkable proof that this code does not violate the safety policy of the host computer. The host computer does not run the given code until it has verified the given proof that the code is safe.

In most current approaches to PCC and TAL [15, 14], the machine-checkable proofs are written in a logic with a built-in understanding of a particular type system. More formally, type constructors appear as primitives of the logic and certain lemmas about these type constructors are built into the verification system. The semantics of the type constructors and the validity of the lemmas concerning them are proved rigorously but without mechnical verification by the designers of the PCC verification system. We will call this type-specialized PCC.

A PCC system must understand not only the language of types, but also the machine language for a particular machine. Necula's PCC systems [15, 7] use a verification-condition generator (VCgen) to derive, for each program, a *verification condition* – a logical formula that if true guarantees the safety of the program. The code producer must prove, and the code consumer must check the proof of, the verification condition. (Both producer and consumer independently run the VCgen to derive the right formula for the given program.)

The VCgen is a fairly large program (23,000 lines of C in the Cedilla Systems implementation [7]) that examines the machine instructions of the program, expands the substitutions of its machine-code Hoare logic, examines the formal parameter declarations to derive function precon-

*This research was supported in part by DARPA award F30602-99-1-0519 and by National Science Foundation grant CCR-9974553.

0-7695-1281-X/01 $10.00 © 2001 IEEE

ditions, and examines result declarations to derive post-conditions. A bug in the VCgen will lead to the wrong formula being proved and checked.

The soundness of a PCC system's typing rules and VCgen can, in principle, be proved as a metatheorem. Human-checked proofs of type systems are almost tractable; the appendices of Necula's thesis [16] and Morrisett et al.'s paper [14] contain such proofs, if not of the actual type systems used in PCC systems, then of their simplified abstractions. But constructing a mechanically-checkable correctness proof of a full VCgen would be a daunting task.

Foundational PCC. Unlike type-specialized PCC, the foundational PCC described by Appel and Felty [3] avoids any commitment to a particular type system and avoids using a VC generator. In foundational PCC the operational semantics of the machine code is defined in a logic that is suitably expressive to serve as a foundation of mathematics. We use higher-order logic with a few axioms of arithmetic, from which it is possible to build up most of modern mathematics. The operational semantics of machine instructions [12] and safety policies [2] are easily defined in higher-order logic. In foundational PCC the code provider must give both the executable code plus a proof in the foundational logic that the code satisfies the consumer's safety policy. The proof must explicitly define, down to the foundations of mathematics, all required concepts and explicitly prove any needed properties of these concepts.

Foundational PCC has two main advantages over type-specialized PCC — it is more flexible and more secure. Foundational PCC is more flexible because the code producer can "explain" a novel type system or safety argument to the code consumer. It is more secure because the trusted base can be smaller: its trusted base consists only of the foundational verification system together with the definition of the machine instruction semantics and the safety policy. A verification system for higher-order logic can be made quite small [10, 17].

In our research project at Princeton University (with the help of many colleages elsewhere) we are building a foundational PCC system, so that we can specify and automatically prove and check the safety of machine-language programs. In this paper I will explain the components of the system.

2 Choice of logic and framework

To do machine-checked proofs, one must first choose a logic and a logical framework in which to manipulate the logic. The logic that we use is Church's higher-order logic with axioms for arithmetic; we represent our logic, and check proofs, in the LF metalogic [10] implemented in the Twelf logical framework [18]. We have chosen LF because it naturally produces proof objects that we can send to a "consumer."

The Twelf system allows us to specify constructors of our object logic. Our object logic has types `tp`; its primitive types are propositions `o` and numbers `num`; there is an `arrow` constructor to build function types, and `pair` to build tuples. For any object-logic type T, object-logic expressions of that type have metalogical type `tm` T. Finally, for any formula A we can talk about proofs of A, which belong to the metalogical type `pf` (A).

```
tp   : type.
tm   : tp -> type.
o: tp.   num: tp.
arrow: tp -> tp -> tp.
   %infix right 14 arrow.
pair: tp -> tp -> tp.
pf   : tm o -> type.
```

We have object-logic constructors `lam` (to construct functions), `@` (to apply a function to an argument, written infix), `imp` (logical implication), and `forall` (universal quantification):

```
lam: (tm T1 -> tm T2) -> tm (T1 arrow T2).
@  : tm (T1 arrow T2) -> tm T1 -> tm T2.
  %infix left 20 @.
imp    : tm o -> tm o -> tm o.
  %infix right 10 imp.
forall : (tm T -> tm o) -> tm o.
```

The trick of using `lam` and `@` to coerce between metalogical functions `tm T1 -> tm T2` and object-logic functions `tm (T1 arrow T2)` is described by Harper, Honsell, and Plotkin [10]. We need object-logic functions so that we can quantify over them using `forall`; that is, the type of `F` in `forall [F]` *`predicate`*`(F)` must be `tm T` for some `T` such as `num arrow num`, but cannot be `tm T1 -> tm T2`.

We have introduction and elimination rules for these constructors (rules for pairing omitted here):

```
beta_e: {P: tm T -> tm o}
         pf(P (lam F @ X)) -> pf(P (F X)).
beta_i: {P: tm T -> tm o}
         pf(P (F X)) -> pf(P (lam F @ X)).

imp_i: (pf A -> pf B) -> pf (A imp B).
imp_e: pf (A imp B) -> pf A -> pf B.
```

```
forall_i:
      ({X:tm T}pf(A X)) -> pf(forall A).
forall_e:
      pf(forall A) -> {X:tm T}pf(A X).
not_not_e: pf ((B imp forall [A] A)
                 imp forall [A] A)
           -> pf B.
```

Our proofs don't need extensionality or the general axiom of choice.

Once we have defined the constructors of the logic, we can define lemmas and new operators as definitions in Twelf:

```
and : tm o -> tm o -> tm o =
  [A][B]
  forall [C] (A imp B imp C) imp C.

  %infix right 12 and.

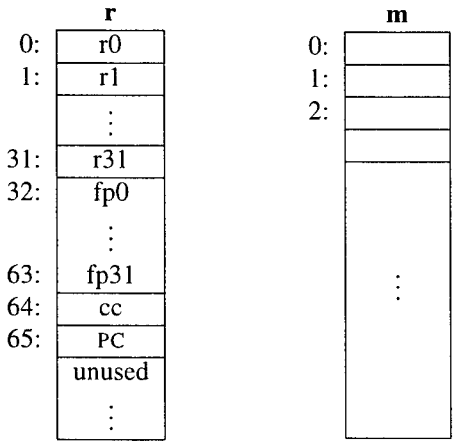
and_i    : pf A -> pf B -> pf (A and B) =
 [p1: pf A][p2: pf B]
 forall_i [c: tm o]
 imp_i [p3] imp_e (imp_e p3 p1) p2.

and_e1   : pf (A and B) -> pf A =
  [p1: pf (A and B)]
  imp_e (forall_e p1 A)
  (imp_i [p2: pf A] imp_i [p3: pf B] p2).
```

Of course, the defined lemmas are checked by machine (the Twelf type checker), and need not be trusted in the same way that the core inference rules are. Our interactive tutorial [1] provides an informal introduction to our object logic.

3 Specifying machine instructions

We start by modeling a specific von Neumann machine, such as the Sparc or the Pentium. A machine state comprises a *register bank* and a *memory*, each of which is a function from integers (addresses) to integers (contents). Every register of the instruction-set architecture (ISA) must be assigned a number in the register bank: the general registers, the floating-point registers, the condition codes, and the program counter. Where the ISA does not specify a number (such as for the PC) we use an arbitrary index:



A single step of the machine is the execution of one instruction. We can specify instruction execution by giving a step relation $(r,m) \mapsto (r',m')$ that describes the relation between the prior state (r,m) and the state (r',m') of the machine after execution.

For example, to describe the instruction $r_1 \leftarrow r_2 + r_3$ we might start by writing,

$$(r,m) \mapsto (r',m') \equiv$$
$$r'(1) = r(2) + r(3) \wedge (\forall x \neq 1.\, r'(x) = r(x)) \wedge m' = m$$

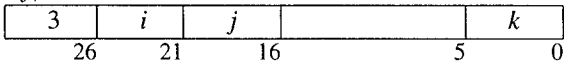
In fact, we can define $\mathrm{add}(i,j,k)$ as this predicate on four arguments (r,m,r',m'):

$$\mathrm{add}(i,j,k) =$$
$$\lambda r,m,r',m'.\ r'(i) = r(j) + r(k) \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m$$

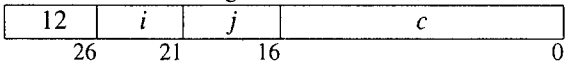
Similarly, we can define the instruction $r_i \leftarrow m[r_j + c]$ as

$$\mathrm{load}(i,j,c) =$$
$$\lambda r,m,r',m'.\ r'(i) = m(r(j)+c) \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m$$

But we must also take account of instruction fetch and decoding. Suppose, for example, that the add instruction is encoded as a 32-bit word, containing a 6-bit field with opcode 3 denoting *add*, a 5-bit field denoting the destination register i, and 5-bit fields denoting the source registers j,k:



The load instruction might be encoded as,



Then we can say that some number w decodes to an instruction *instr* iff,

$$
\begin{aligned}
&\text{decode}(w, instr) \equiv \\
&\quad (\exists i, j, k. \\
&\qquad 0 \le i < 2^5 \ \wedge\ 0 \le j < 2^5 \ \wedge\ 0 \le k < 2^5 \ \wedge \\
&\qquad w = 3 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + k \cdot 2^0 \ \wedge \\
&\qquad instr = \text{add}(i, j, k)) \\
&\quad \vee (\exists i, j, c. \\
&\qquad 0 \le i < 2^5 \ \wedge\ 0 \le j < 2^5 \ \wedge\ 0 \le c < 2^{16} \ \wedge \\
&\qquad w = 12 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + c \cdot 2^0 \ \wedge \\
&\qquad instr = \text{load}(i, j, \text{sign-extend}(c))) \\
&\quad \vee \ \ldots
\end{aligned}
$$

with the ellipsis denoting the many other instructions of the machine, which must also be specified in this formula.

Neophytos Michael and I have shown [12] how to scale this idea up to the instruction set of a real machine. Real machines have large but semiregular instruction sets; instead of a single global disjunction, the decode relation can be factored into operands, addressing modes, and so on. Real machines don't use integer arithmetic, they use modular arithmetic, which can itself be specified in our higher-order logic. Some real machines have multiple program counters (e.g., Sparc) or variable-length instructions (e.g., Pentium), and these can also be accommodated.

Our description of the decode relation is heavily factored by higher-order predicates (this would not be possible without higher-order logic). We have specified the execution behavior of a large subset of the Sparc architecture (without register windows or floating-point). For PCC, it is sufficient to specify a subset of the machine architecture; any unspecified instruction will be treated by the safety policy as illegal, which may be inconvenient for compilers that want to generate that instruction, but which cannot compromise safety.

Our Sparc specification has two components, a "syntactic" part (the decode relation) and a semantic part (the definitions of *add*, *load*, etc.). The syntactic part is derived from a 151-line specification written in the SLED language of the New Jersey Machine-Code Toolkit [19]; our translator expands this to 1035 lines of higher-order logic, as represented in Twelf; but we believe that a more concise and readable translation would produce only 500–600 lines. The semantic part is about 600 lines of logic, including the definition of modular arithmetic.

4 Specifying safety

Our step relation $(r, m) \mapsto (r', m')$ is deliberately partial; some states have no successor state. In these states the program counter $r(\text{PC})$ points to an illegal instruction. Now we will proceed to make it even more partial, by defining as illegal those instructions that violate our safety policy.

For example, suppose we wish to specify a safety policy that "only *readable* addresses will be loaded," where the predicate *readable* is given some suitable definion such as

$$\text{readable}(x) = 0 \le x < 1000$$

(see Appel and Felten [2] for descriptions of security policies that are more interesting than this one).

We can add a new conjunct to the semantics of the *load* instruction,

$$
\begin{aligned}
&\text{load}(i, j, c) \ = \\
&\quad \lambda r, m, r', m'.\ r'(i) = m(r(j) + c) \\
&\qquad\qquad \wedge\ \text{readable}(r(j) + c) \\
&\qquad\qquad \wedge\ (\forall x \neq i.\, r'(x) = r(x)) \ \wedge\ m' = m.
\end{aligned}
$$

Now, in a machine state where the program counter points to a load instruction that violates the safety policy, our step relation $\mapsto$ does not relate this state to any successor state (even though the real machine "knows how" to execute it).

Using this partial step relation, we can define safety; a given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state:

$$
\begin{aligned}
&\text{safe-state}(r, m) \ = \\
&\quad \forall r', m'.\ (r, m \mapsto^* r', m') \ \Rightarrow\ \exists r'', m''.\ r', m' \mapsto r'', m''
\end{aligned}
$$

A program is just a sequence of integers (representing machine instructions); we say that a program p is loaded at a location *start* in memory m if

$$\text{loaded}(p, m, start) \ = \ \forall i \in \text{dom}(p).\ m(i + start) = p(i)$$

Finally (assuming that programs are written in position-independent code), a program is *safe* if, no matter where we load it in memory, we get a safe state:

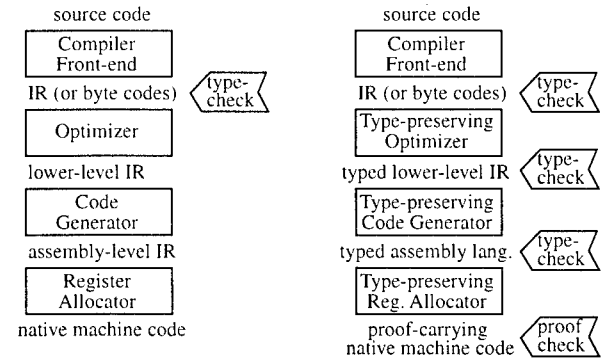
$$
\begin{aligned}
&\text{safe}(p) \ = \\
&\quad \forall r, m, start.\ \ \text{loaded}(p, m, start) \ \wedge\ r(\text{PC}) = start \ \Rightarrow \\
&\qquad\qquad \text{safe-state}(r, m)
\end{aligned}
$$

The important thing to notice about this formulation is that *there is no verification-condition generator*. The syntax and semantics of machine instructions, implicit in a VCgen, have been made explicit – and much more concise – in the step relation. But the Hoare logic of machine instructions and typing rules for function parameters, also implicit in a VCgen, must now be proved as lemmas – about which more later.

5 Proving safety

In a sufficiently expressive logic, as we all know, proving theorems can be a great deal more difficult than merely stating them – and higher-order logic is certainly expressive. For guidance in proving safety of machine-language programs we should not particularly look to previous work in formal verification of program correctness. Instead, we should think more of type checking: automatic proofs of decidable safety properties of programs.

The key advances that makes it possible to generate proofs automatically are *typed intermediate languages* [11] and *typed assembly language* [14]. Whereas conventional compilers type-check the source program, then throw away the types (using the lambda-calculus principle of *erasure*) and then transform the program through progressively lower-level intermediate representations until they reach assembly language and then machine language, a type-preserving compiler uses typed intermediate languages at each level. If the program type-checks at a low level, then it is safe, regardless of whether the previous (higher-level) compiler phases might be buggy on some inputs. As the program is analyzed into smaller pieces at the lower levels, the type systems become progressively more complex, but the type theory of the 1990's is up to the job of engineering the type systems.



Conventional Compiler **Type-preserving Compiler**

TAL was originally designed to be used in a certifying compiler, but one that certifies the assembly code and uses a trusted assembler to translate to machine code. But we can use TAL to help generate proofs in a PCC system that directly verifies the machine code. In such a system, the proofs are typically by induction, with induction hypotheses such as, "whenever the program-counter reaches location l, the register 3 will be a pointer to a pair of integers." These local invariants can be generated from the TAL formulation of the program, but in a PCC system they can be checked in machine code without needing to trust the assembler.

Typing rules for machine language. In important insight in the development of PCC is that one can write type-inference rules for machine language and machine states. For example, Necula [15] used rules such as

$$\frac{m \vdash x : \tau_1 \times \tau_2}{m \vdash m(x) : \tau_1 \wedge m(x+1) : \tau_2}$$

meaning that if x has type $\tau_1 \times \tau_2$ in memory m – meaning that it is a pointer to a boxed pair – then the contents of location x will have type τ_1 and the contents of location $x+1$ will have type τ_2.

Proofs of safety in PCC use the local induction hypotheses at each point in the program to prove that the program is typable. This implies, by a type-soundness argument, that the program is therefore safe.

If the type system is given by syntactic inference rules, the proof of type soundness is typically done by syntactic subject reduction – one proves that each step of computation preserves typability and that typable states are safe. The proof involves structural induction over typing derivations. In conventional PCC, this proof is done in the metatheory, by humans.

In foundational PCC we wish to include the type-soundness proof inside the proof that is transmitted to the code consumer because (1) it's more secure to avoid reliance on human-checked proofs and (2) that way we avoid restricting the protocol to a single type system. But in order to do a foundational subject-reduction theorem, we would need to build up the mathematical machinery to manipulate typing derivations as syntactic objects, all represented inside our logic using foundational mathematical concepts – sets, pairs, and functions. We would need to do case analyses over the different ways that a given type judgement might be derived. While this can all be done, we take a different approach to proving that typability implies safety.

We take a semantic approach. In a semantic proof one assigns a meaning (a semantic truth value) to type judgements. One then proves that if a type judgement is true then the typed machine state is safe. One further proves that the type inference rules are sound, i.e., if the premises are true then the conclusion is true. This ensures that derivable type judgements are true and hence typable machine states are safe.

The semantic approach avoids formalizing syntactic type expressions. Instead, one formalizes a type as a set of semantic values. One defines the operator $\times$ as a function taking two sets as arguments and returning a set. The

above type inference rule for pair projection can then be replaced by the following semantic lemma in the foundational proof:

$$\frac{\models x :_m \tau_1 \times \tau_2}{\models m(x) :_m \tau_1 \wedge m(x+1) :_m \tau_2}$$

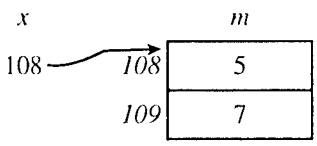
Although the two forms of the application type-inference rule look very similar they are actually significantly different. In the second rule τ_1 and τ_2 range over semantic sets rather than type expressions. The relation $\models$ in the second version is defined directly in terms of a semantics for assertions of the form $x :_m \tau$. The second "rule" is actually a lemma to be proved while the first rule is simply a part of the definition of the syntactic relation $\vdash$. For the purposes of foundational PCC, we view the semantic proofs as preferable to syntactic subject-reduction proofs because they lead to shorter and more manageable foundational proofs. The semantic approach avoids the need for any formalization of type expressions and avoids the formalization of proofs or derivations of type judgements involving type expressions.

5.1 Semantic models of types

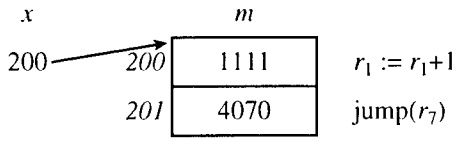
Building semantic models for type systems is interesting and nontrivial. In a first attempt, Amy Felty and I [3] were able to model a pure-functional (immutable datatypes) call-by-value language with records, address arithmetic, polymorphism and abstract types, union and intersection types, continuations and function pointers, and covariant recursive types.

Our simplest semantics is set-theoretic: a type is a set of values. But what is a value? It is not a syntactic construct, as in lambda-calculus; on a von Neumann machine we wish to use a more natural representation of values that corresponds to the way procedures and data structures are represented in practice. This way, our type theory can match reality without a layer of simulation in between. We can represent a value as a pair (m,x), where m is a memory and x is an integer (typically representing an address).

To represent a pointer data structure that occupies a certain portion of the machine's memory, we let x be the root address of that structure. For example, the boxed pair of integers $\langle 5,7\rangle$ represented at address 108 would be represented as the value $(\{108 \mapsto 5, 109 \mapsto 7\}, 108)$.



To represent a function value, we let x be the entry address of the function; here is the function $f(x) = x+1$, assuming that arguments and return results are passed in register 1:



This model of values would be sufficient in a semantics of statically allocated data structures, but to have dynamic heap allocation we must be able to indicate the set a of allocated addresses, such that any modification of memory outside the allocated set will not disturb already allocated values. A *state* is a pair (a,m), and a value is a pair $((a,m),x)$ of state and root-pointer. The alloc set a is virtual: it is not directly represented at run time, but is existentially quantified.

Limitations. In the resulting semantics [3] we could model heap allocation, but we could not model mutable record-fields; and though our type system could describe

```
datatype 'a list = nil
                 | :: of 'a * 'a list
```

we could not handle recursions where the type being defined occurs in a negative (contravariant) position, as in

```
datatype exp = APP of exp * exp
             | LAM of [exp] -> exp
```

where the boxed occurrence of `exp` is a negative occurrence. Contravariant recursion is occasionally useful in ML, but it is the very essence of object-oriented programming, so these limitations (no mutable fields, no contravariant recursion) are quite restrictive.

5.2 Indexed model of recursive types

In more recent work, David McAllester and I have shown how to make an "indexed" semantic model that can describe contravariant recursive types [4]. Instead of saying that a type is a set of values, we say that it is a set of pairs $\langle k,v\rangle$ where k is an approximation index and v is a value. The judgement $\langle k,v\rangle \in \tau$ means, "v approximately has type τ, and any program that runs for fewer than k instructions can't tell the difference." The indices k allow the construction of a well founded recursion, even when modeling contravariant recursive types.

The type system works both for von Neumann machines and for λ-calculus; here I will illustrate the latter. We define a *type* as a set of pairs $\langle k,v\rangle$ where k is a non-negative integer and v is a value and where the set τ is

such that if $\langle k, v\rangle \in \tau$ and $0 \leq j \leq k$ then $\langle j, v\rangle \stackrel{\cdot}{\in} \tau$. For any closed expression e and type τ we write $e :_k \tau$ if e is safe for k steps and if whenever $e \mapsto^j v$ for some value v with $j < k$ we have $\langle k-j, v\rangle \in \tau$; that is,

$$e :_k \tau \equiv \forall j \forall e'.\ 0 \leq j < k \wedge e \mapsto^j e' \wedge \mathrm{nf}(e') \Rightarrow \langle k-j, e'\rangle \in \tau$$

where $\mathrm{nf}(e')$ means that e' is a normal form — has no successor in the call-by-value small-step evaluation relation.

We start with definitions for the sets that represent the types:

$$\begin{array}{rcl} \bot & \equiv & \{\} \\ \top & \equiv & \{\langle k, v\rangle \mid k \geq 0\} \\ \textbf{int} & \equiv & \{\langle k, \mathbf{0}\rangle, \langle k, \mathbf{1}\rangle, \ldots \mid k \geq 0\} \\ \tau_1 \times \tau_2 & \equiv & \{\langle k, (v_1, v_2)\rangle \mid \forall j < k.\ \langle j, v_1\rangle \in \tau_1 \wedge \langle j, v_2\rangle \in \tau_2\} \\ \sigma \to \tau & \equiv & \{\langle k, \lambda x.e\rangle \mid \forall j < k \forall v.\ \langle j, v\rangle \in \sigma \Rightarrow e[v/x] :_j \tau\} \\ \mu F & \equiv & \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\} \end{array}$$

Next we define what is meant by a typing judgement. Given a mapping Γ from variables to types, we write $\Gamma \vDash_k e : \alpha$ to mean that

$$\forall \sigma.\, \sigma :_k \Gamma \Rightarrow \sigma(e) :_k \alpha$$

where $\sigma(e)$ is the result of replacing the free variables in e with their values under substitution σ. To drop the index k, we define

$$\Gamma \vDash e : \alpha \equiv \forall k.\, \Gamma \vDash_k e : \alpha$$

Soundness theorem: It is trivial to prove from these definitions that if $\vDash e : \alpha$ and $e \mapsto^* e'$ then e' is not stuck, that is, $e' \mapsto e''$.

Well founded type constructors. We define the notion of a well founded type constructor. Here I will not give the formal definition, but state the informal property that if F is well founded and $x : F(\tau)$, then to extract from x a value of type τ, or to apply x to a value of type τ, must take at least one execution step. The constructors $\times$ and $\to$ are well founded.

Typing rules. Proofs of theorems such as the following are not too lengthy:

$$\frac{\Gamma \vDash \pi_1(e) : \tau_1 \qquad \Gamma \vDash \pi_2(e) : \tau_2}{\Gamma \vDash e : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vDash e : \tau_1 \times \tau_2}{\Gamma \vDash \pi_1(e) : \tau_1}$$

$$\frac{\Gamma \vDash e_1 : \alpha \to \beta \qquad \Gamma \vDash e_2 : \alpha}{\Gamma \vDash e_1 e_2 : \beta}$$

Finally, for any well founded type-constructor F, we have equirecursive types: $\mu F = F(\mu F)$.

Our paper [4] proves all these theorems and shows the extension of the result to types and values on von Neumann machines.

5.3 Mutable fields

Our work on mutable fields is still in a preliminary stage. Amal Ahmed, Roberto Virga, and I are investigating the following idea. Our semantics of immutable fields viewed a "state" as a pair (a, m) of a memory m and a set a of allocated addresses. To allow for the update of existing values, we enhance a to become a finite map from locations to types. The type $a(l)$ at some location l specifies what kinds of updates at that location will preserve all existing typing judgements. Then, as before, a type is a predicate on states (a, m) and root-pointers x of type integer. In our object logic, we would write the types of these logical objects as,

$$\begin{array}{rcl} \mathit{allocset} & = & \mathit{num} \xrightarrow{\text{fin}} \mathit{type} \\ \mathit{value} & = & \mathit{allocset} \times \mathit{memory} \times \mathit{num} \\ \mathit{type} & = & \mathit{num} \times \mathit{value} \ \to \ o \end{array}$$

The astute reader will notice that the metalogical type of "type" is recursive, and in a way that has an inconsistent cardinality: the set of types must be bigger than itself. This problem had us stumped for over a year, but we now have a tentative solution that replaces the type (in the allocset) with the Gödel number of a type. We hope to report on this result soon; we are delayed by our general practice of machine-checking our proofs in Twelf before submitting papers for publication, which in this case has saved us from some embarrassment.

5.4 Typed machine language

Morrisett's typed assembly language [14] is at too high a level to do proof-carrying code directly. Kedar Swadi, Gang Tan, Roberto Virga, and I have been designing a lower-level representation, called *typed machine language*, that will serve as the interface between compilers and our prover. In fact, we hope that a clean enough definition of this language will shift most of the work from the prover to the compiler's type-checker.

In order to avoid overspecializing the typed machine language (TML) with language-specific constructs such as records and disjoint-union variants, our TML will use very low-level typing primitives such as union types, intersection types, offset (address-arithmetic) types, and de-

pendent types. This will make type-checking of TML difficult; we will need to assume that each compiler will have a source language with a decidable type system, and that translation of terms (and types) will yield a witness to the type-checking of the resultant TML representation.

Abstract machine instructions. One can view machine instructions at many levels of abstraction:

1. At the lowest level, an instruction is just an integer, an opcode encoding.

2. At the next level, it implements a relation on raw machine states $(r,m) \mapsto (r',m')$.

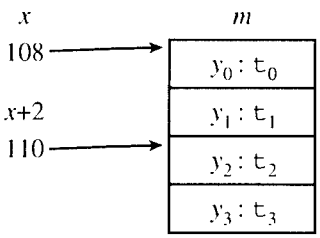
3. At a higher level, we can say that the Sparc *add* instruction implements a machine-independent notion of *add*, and similarly for other instruction.

4. Then we can view *add* as manipulating not just registers, but local variables (which may be implemented in registers or in the activation record).

5. We can view this instruction as one of various typed instructions on typed values; in the usual view, *add* has type $\text{int} \times \text{int} \rightarrow \text{int}$, but the address-arithmetic *add* has type

   $$(\tau_0 \times \tau_1 \times \ldots \times \tau_n) \times \text{const}(i) \rightarrow (\tau_i \times \tau_{i-1} \times \ldots \times \tau_n)$$

   for any i.

6. Finally, we can specialize this typed *add* to the particular context where some instance of it appears, for example by instantiating the i, n, and τ_i in the previous example.

Abstraction level 1 is used in the statement of the theorem (safety of a machine-language program p). Abstraction level 5 is implicitly used in conventional proof-carrying code [15]. Our ongoing research involves finding semantic models for each of these levels, and then proving lemmas that can convert between assertions at the different levels.

Hoare logic. In reasoning about machine instructions at a higher level of abstraction, notions from Hoare logic are useful: preconditions, postconditions, and substition. Without adding any new axioms, we can define a notion of predicates on states to serve as preconditions and postconditions, and substitution as a relation on predicates. But this can rapidly become inefficient, leading to proofs that are quadratic or exponential in size. Kedar Swadi, Roberto Virga, and I have taken some steps in lemmatizing substitution so that proofs don't blow up [5]; interesting related work has been done in Compaq SRC's extended static checker [9].

Software engineering practices. We define all of these abstraction levels in order to modularize our proofs. Since our approach to PCC shifts most of the work to the human prover of static, machine-checkable lemmas about the programming language's type system, we find it imperative to use the same software engineering practices in implementing proofs as are used in building any large system. The three most important practices are (1) abstraction and modularity, (2) abstraction and modularity, and (3) abstraction and modularity. At present, we have about thirty thousand lines of machine-checked proofs, and we would not be able to build and maintain the proofs without a well designed modularization.

6 Pruning the runtime system

Just as bugs in the compiler (of a conventional system) or the proof checker (of a PCC system) can create security holes, so can bugs in the runtime system: the garbage collector, debugger, marshaller/unmarshaller, and other components. An important part of research in Foundational PCC is to move components from the runtime system to the type-checkable user code. Then, any bugs in such components will either be detected by type-checking (or proof-checking), or will be type-safe bugs that may cause incorrect behavior but not insecure behavior.

Garbage collectors do two strange things that have made them difficult to express in a type-safe language: they allocate and deallocate arenas of memory containing many objects of different types, and they traverse (and copy) objects of arbitrary user-chosen types. Daniel Wang has developed a solution to these problems [22], based on the motto,

$$\text{Garbage collection} = \text{Regions} + \text{Intensional types}.$$

That is, the region calculus of Tofte and Talpin [20] can

be applied to the problem of garbage collection, as noticed in important recent work by Walker, Crary, and Morrisett [21]; to traverse objects of unknown type, the intensional type calculi of originally developed by Harper and Morrisett [11] can be applied. Wang's work covers the region operators and management of pointer sharing; related work by Monnier, Saha, and Shao [13] covers the intensional type system.

Other potentially unsafe parts of the runtime system are ad hoc implementations of *polytypic* functions – those that work by induction over the structure of data types – such as polymorphic equality testers, debuggers, and marshallers (a.k.a. serializers or picklers). Juan Chen and I have developed an implementation of polytypic primitives as a transformation on the typed intermediate representation in the SML/NJ compiler [6]. Like the λ_R transformation of Crary and Weirich [8] it allows these polytypic functions to be typechecked, but unlike their calculus, ours does not require dependent types in the typed intermediate language and is thus simpler to implement.

7 Conclusion

Our goal is to reduce the size of the trusted computing base of systems that run machine code from untrusted sources. This is an engineering challenge that requires work on many fronts. We are fortunate that during the last two decades, many talented scientists have built the mathematical infrastructure we need – the theory and implementation of logical frameworks and automated theorem provers, type theory and type systems, compilation and memory management, and programming language design. The time is ripe to apply all of these advances as engineering tools in the construction of safe systems.

References

[1] Andrew W. Appel. Hints on proving theorems in Twelf. www.cs.princeton.edu/~appel/twelf-tutorial, February 2000.

[2] Andrew W. Appel and Edward W. Felten. Models for security policies in proof-carrying code. Technical Report TR-636-01, Princeton University, March 2001.

[3] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, January 2000.

[4] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. Technical Report TR-629-00, Princeton University, October 2000.

[5] Andrew W. Appel, Kedar N. Swadi, and Roberto Virga. Efficient substitution in hoare logic expressions. In *4th International Workshop on Higher-Order Operational Techniques in Semantics (HOOTS 2000)*. Elsevier, September 2000. Electronic Notes in Theoretical Computer Science 41(3).

[6] Juan Chen and Andrew W. Appel. Dictionary passing for polytypic polymorphism. Technical Report CS-TR-635-01, Princeton University, March 2001.

[7] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[8] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM SIGPLAN International Conference on Functional Programming Languages*, September 1999.

[9] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205. ACM Press, January 2001.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[11] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[12] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[13] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, page to appear, June 2001.

[14] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[15] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[16] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[17] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[18] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[19] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Trans. on Programming Languages and Systems*, 19(3):492–524, May 1997.

[20] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-first ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[21] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Trans. on Programming Languages and Systems*, 22(4):701–771, July 2000.

[22] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–178. ACM Press, January 2001.

Session 7

Intuitionistic Linear Logic and Partial Correctness

Dexter Kozen
Cornell University
kozen@cs.cornell.edu

Jerzy Tiuryn
Warsaw University
tiuryn@mimuw.edu.pl

Abstract

We formulate a Gentzen-style sequent calculus for partial correctness that subsumes propositional Hoare Logic. The system is a noncommutative intuitionistic Linear Logic. We prove soundness and completeness over relational and trace models. As a corollary we obtain a complete sequent calculus for inclusion and equivalence of regular expressions.

1 Introduction

In formulating logics for program verification such as Hoare Logic (HL), Dynamic Logic (DL), or Kleene Algebra with Tests (KAT), it is tempting to treat tests and correctness assertions as a uniform syntactic category. This temptation is best resisted: although both are classes of assertions, they have quite different characteristics. *Tests* are local assertions whose truth is determined by the current state of execution. They are normally immediately decidable. The assertion $x \geq 0$, where x is a program variable, is an example of such a test. Tests occur in all modern programming languages as part of conditional expressions and looping constructs. *Correctness assertions*, on the other hand, are statements about the global behavior of a program, such as partial correctness or halting. They are typically much richer in expressive power than tests and undecidable in general.

DL does not distinguish between these two categories of assertions. The two are freely mixed, and both are treated classically. For this reason, the resulting system is unnecessarily complex for its purposes. The rich-test version of DL, in which one can convert an arbitrary correctness assertion to a test using the operator ?, is Π^1_1-complete (see [9]). Even with systems that do make the distinction, such as KAT, care must be taken not to inadvertently treat global properties as local; doing so can lead to anomalies such as the Dead Variable Paradox [13].

One major distinguishing factor between tests and correctness assertions that may not be immediately apparent is that the former are classical in nature, whereas the latter are intuitionistic. For example, the DL axiom

$$[p][q]b \quad \equiv \quad [p\,;q]b$$

can be regarded as a noncommutative version of the intuitionistic currying rule

$$p \to q \to b \quad \equiv \quad p \wedge q \to b.$$

Gödel [8] first observed the strong connection between modal and intuitionistic logic, foreshadowing Kripke's formulation of similar state-based semantics for these logics [16, 17] (see [1]). Kripke models also form the basis of the standard semantics of DL(see [9]), although as mentioned, DL does not realize the intuitionistic nature of partial correctness.

In this paper we give a Gentzen-style sequent calculus S that clearly separates partial correctness reasoning into its classical and intuitionistic parts. In Section 4, where we introduce the system, we will explain why we view partial correctness reasoning in S as intuitionistic rather than classical. System S has the flavor of a noncommutative intuitionistic Linear Logic and is in some ways related to a system of Girard [6, 7]. It is linear because expressions cannot be indiscriminately duplicated or eliminated.

The system does not contain any contraction rules. The linear implication operator takes only programs as left argument, while arbitrary partial correctness formulas can occur on the right. There is a very limited way in which the weakening rule for programs can be used—programs can be inserted only at front of an environment. There is a cocontraction rule: a program of the form p^+ already present in the environment can be duplicated. Troelstra [20, p. 25] remarks that contraction has more dramatic proof theoretic consequences than weakening when added to Linear Logic.

We give relational and trace semantics for this logic and show how the logic captures partial correctness. We then prove soundness and completeness over both classes of models. As a corollary we obtain a complete sequent calculus for inclusion and equivalence of regular expressions.

We mention that our two equivalent semantics of Section 3 are both special cases of a more general approach to the

0-7695-1281-X/01 $10.00 © 2001 IEEE

semantics of noncommutative Linear Logic via quantales [21]. We restrict our attention to two special kinds of quantales: sets of traces and binary relations. Our completeness result is thus stronger than it would be for the more general semantics based on arbitrary quantales.

2 Syntax

The syntax of S comprises several syntactic categories. These will require some intuitive explanation, which we defer until after the formal definition. In particular we distinguish between two kinds of propositions, which we call *tests* and *formulas*.

tests	$b, c, d \ldots$	$b ::= \langle\text{atomic tests}\rangle \mid \bot \mid b \to c$
programs	$p, q, r, \ldots$	$p ::= \langle\text{atomic programs}\rangle \mid b \mid p \sqcup q \mid p \otimes q \mid p^+$
formulas	$\varphi, \psi, \ldots$	$\varphi ::= b \mid p \to \varphi$
environments	$\Gamma, \Delta, \ldots$	$\Gamma ::= \varepsilon \mid \Gamma, p \mid \Gamma, \varphi$
sequents	$\Gamma \vdash \varphi$	

In the above grammar, $\to$ is called *linear implication*, $\otimes$ is a noncommutative multiplicative connective called *tensor*, $\sqcup$ is a commutative additive connective called *disjunction*, and $^+$ is a unary operation called *positive iteration*. We use brackets where necessary to ensure unique readability. We abbreviate $b \to \bot$ by $\overline{b}$, $\overline{\bot}$ by $\mathbf{1}$, $p \otimes q$ by pq, and $\mathbf{1} \sqcup p^+$ by p^*.

Several formalisms, such as PDL [5] and KAT [14], are based on * rather than $^+$. We can freely move between the two languages since * and $^+$ are mutually definable:

$$p^* = \mathbf{1} \sqcup p^+ \qquad p^+ = pp^*.$$

For this reason, models for one language can be viewed as models for the other.

We base S on $^+$ instead of * because the resulting deductive system is cleaner—it contains no contraction rule[1]. This is perhaps due to the fact that $^+$ can be viewed as a more primitive operation than *.

A *test* is either an atomic test, the symbol $\bot$ representing falsity, or an expression $b \to c$ representing classical implication, where b and c are tests. We use the symbols $b, c, d, \ldots$ exclusively to stand for tests. The set of all tests is denoted $\mathcal{B}$. The sequent calculus to be presented in Section 4 will encode classical propositional logic for tests.

A *program* is either an atomic program, a test, or an expression $p \sqcup q$, $p \otimes q$, or p^+, where p and q are programs. We use the symbols $p, q, r, \ldots$ exclusively to stand for programs. The set of all programs is denoted $\mathcal{P}$. As in PDL [5], the program operators can be used to construct conventional procedural programming constructs such as conditional tests and while loops.

A *formula* is either a test or an expression $p \to \varphi$, read "after p, φ," where p is a program and φ is a formula. Intuitively, the meaning is similar to the DL modal construct $[p]\varphi$. The operator $\to$ associates to the right. We use the symbols $\varphi, \psi, \ldots$ to stand for formulas.

Environments are denoted $\Gamma, \Delta, \ldots$. An environment is a (possibly empty) sequence of programs and formulas. The empty environment is denoted ε. Intuitively, an environment describes a previous computation that has led to the current state.

Sequents are of the form $\Gamma \vdash \varphi$, where Γ is an environment and φ is a formula. We write $\vdash \varphi$ for $\varepsilon \vdash \varphi$. Intuitively, the meaning of $\Gamma \vdash \varphi$ is similar to the DL assertion $[\Gamma]\varphi$, where we think of the environment $\Gamma = \ldots, p, \ldots, \psi, \ldots$ as the rich-test program $\cdots; p; \cdots; \psi?; \cdots$ of DL.

The partial correctness assertion $\{b\}\, p\, \{c\}$ of HL is encoded by the formula $b \to p \to c$. The Hoare-style rule

$$\frac{\{b_1\}\, p_1\, \{c_1\},\ \ldots,\ \{b_n\}\, p_n\, \{c_n\}}{\{b\}\, p\, \{c\}}$$

is encoded by the sequent

$$b_1 \to p_1 \to c_1, \ldots, b_n \to p_n \to c_n \vdash b \to p \to c.$$

It follows from Theorem 6.1 that all relationally valid rules of this form are derivable; this is false for HL (see [11, 15]).

3 Semantics

3.1 Guarded Strings

Guarded strings over P, B were introduced in [14]. We review the definition here.

Let $\mathsf{B} = \{b_1, \ldots, b_k\}$ and $\mathsf{P} = \{p_1, \ldots, p_m\}$ be fixed finite sets of atomic tests and atomic programs, respectively. An *atom* of B is a program $\ell_1 \cdots \ell_k$ such that ℓ_i is either b_i or $\overline{b}_i$. We require for technical reasons that the ℓ_i occur in this order. An atom represents a minimal nonzero element of the free Boolean algebra on B. We denote by $\mathcal{A}_\mathsf{B}$ the set of all atoms of B. For an atom α and a test b, we write $\alpha \le b$ if $\alpha \to b$ is a classical propositional tautology.

A *guarded string* is a sequence

$$\sigma = \alpha_0 q_1 \alpha_1 \cdots \alpha_{n-1} q_n \alpha_n,$$

where $n \ge 0$, each $\alpha_i \in \mathcal{A}_\mathsf{B}$, and $q_i \in \mathsf{P}$. We define $\mathbf{first}(\sigma) = \alpha_0$ and $\mathbf{last}(\sigma) = \alpha_n$.

If $\mathbf{last}(\sigma) = \mathbf{first}(\tau)$, we can form the *fusion product* $\sigma\tau$ by concatenating σ and τ, omitting the extra copy of

[1] In fact, one of the natural rules for * is a co-weakening rule, which is a strong form of a contraction rule.

$\mathbf{last}(\sigma) = \mathbf{first}(\tau)$ in between. For example, if $\sigma = \alpha p \beta$ and $\tau = \beta q \gamma$, then $\sigma\tau = \alpha p \beta q \gamma$. If $\mathbf{last}(\sigma) \neq \mathbf{first}(\tau)$, then $\sigma\tau$ does not exist.

For sets X, Y of guarded strings, define

$$X \circ Y \stackrel{\text{def}}{=} \{\sigma\tau \mid \sigma \in X,\ \tau \in Y,\ \sigma\tau \text{ exists}\}$$
$$X^0 \stackrel{\text{def}}{=} \mathcal{A}_{\mathsf{B}}, \qquad X^{n+1} \stackrel{\text{def}}{=} X \circ X^n.$$

Although fusion product is a partial operation on guarded strings, the operation $\circ$ is a total operation on sets of guarded strings. If there is no existing fusion product between an element of X and an element of Y, then $X \circ Y = \varnothing$.

Each program p denotes a set $GS(p)$ of guarded strings:

$$\begin{aligned}
GS(p) &\stackrel{\text{def}}{=} \{\alpha p \beta \mid \alpha, \beta \in \mathcal{A}_{\mathsf{B}}\}, \quad p \text{ atomic} \\
GS(b) &\stackrel{\text{def}}{=} \{\alpha \in \mathcal{A}_{\mathsf{B}} \mid \alpha \leq b\}, \quad b \text{ a test} \\
GS(p \sqcup q) &\stackrel{\text{def}}{=} GS(p) \cup GS(q) \\
GS(p \otimes q) &\stackrel{\text{def}}{=} GS(p) \circ GS(q) \\
GS(p^+) &\stackrel{\text{def}}{=} \bigcup_{n \geq 1} GS(p)^n.
\end{aligned}$$

It follows that $GS(p^*) = \bigcup_{n \geq 0} GS(p)^n$. A guarded string σ is itself a program, and $GS(\sigma) = \{\sigma\}$.

A set of guarded strings over P, B is *regular* if it is $GS(p)$ for some program p. The regular sets of guarded strings form the free Kleene algebra with tests on generators P, B [14]; in other words, $GS(p) = GS(q)$ iff $p = q$ is a theorem of KAT.

Lemma 3.1 *The regular sets of guarded strings are closed under the Boolean operations.*

Proof. Closure under $\varnothing$ and union are explicit by means of the constructs $\bot$ and $\sqcup$. It was shown in [14] that for any program p, there is an equivalent program $\widehat{p}$ such that $GS(p) = GS(\widehat{p}) = R(\widehat{p})$, where $R(\widehat{p})$ is the regular set of strings over the alphabet $\mathsf{P} \cup \mathsf{B} \cup \{\bar{b} \mid b \in \mathsf{B}\}$ denoted by $\widehat{p}$ under the usual interpretation of regular expressions. For example, if $w = (p_1 \sqcup \cdots \sqcup p_m)^*$, we might take $\widehat{w} = (b(p_1 \sqcup \cdots \sqcup p_m))^* b$, where $b = (b_1 \sqcup \bar{b}_1) \cdots (b_k \sqcup \bar{b}_k)$. The set $GS(w) = GS(\widehat{w}) = R(\widehat{w})$ is the set of all guarded strings.

It remains to show closure under complement; closure under intersection follows by the De Morgan laws. Let p' be an expression such that $R(p') = R(\widehat{w}) - R(\widehat{p})$. The expression p' exists since the regular sets of strings over $\mathsf{P} \cup \mathsf{B} \cup \{\bar{b} \mid b \in \mathsf{B}\}$ are closed under the Boolean operations. Then $R(p')$ is a set of guarded strings since $R(\widehat{w})$ is, and

$$GS(p') = R(p') = R(\widehat{w}) - R(\widehat{p}) = GS(w) - GS(p).$$

∎

3.2 Trace Models

Traces are similar to guarded strings but more general. They are defined in terms of Kripke frames. A *Kripke frame* over P, B is a structure $(K,\ \mathfrak{m}_K)$, where

$$\mathfrak{m}_K : \mathsf{P} \to 2^{K \times K} \qquad \mathfrak{m}_K : \mathsf{B} \to 2^K.$$

Elements of K are called *states*. A *trace* in K is a sequence of the form $s_0 q_1 s_1 \cdots s_{n-1} q_n s_n$, where $n \geq 0$, $s_i \in K$, $q_i \in \mathsf{P}$, and $(s_i, s_{i+1}) \in \mathfrak{m}_K(q_{i+1})$ for $0 \leq i \leq n-1$. The first and last states of σ are denoted $\mathbf{first}(\sigma)$ and $\mathbf{last}(\sigma)$, respectively. If $\mathbf{last}(\sigma) = \mathbf{first}(\tau)$, we can fuse σ and τ to get the trace $\sigma\tau$. If $\mathbf{last}(\sigma) \neq \mathbf{first}(\tau)$ then $\sigma\tau$ does not exist. A trace $s_0 q_1 s_1 \cdots s_{n-1} q_n s_n$ is *acyclic* if the s_i are distinct. The model K is *acyclic* if all traces are acyclic. It is no loss of generality to restrict attention to acyclic models; every model is equivalent to an acyclic model obtained by "unwinding" the original model (see [9, p. 132] for an explicit construction).

If X and Y are sets of traces, define

$$X \circ Y \stackrel{\text{def}}{=} \{\sigma\tau \mid \sigma \in X,\ \tau \in Y,\ \sigma\tau \text{ exists}\}$$
$$X^0 \stackrel{\text{def}}{=} K, \qquad X^{n+1} \stackrel{\text{def}}{=} X \circ X^n.$$

Tests, programs, formulas, and environments are interpreted as sets of traces according to the following inductive definition:

$$\begin{aligned}
[\![p]\!]_K &\stackrel{\text{def}}{=} \{spt \mid (s,t) \in \mathfrak{m}_K(p)\}, \quad p \text{ atomic} \\
[\![b]\!]_K &\stackrel{\text{def}}{=} \mathfrak{m}_K(b), \quad b \text{ atomic} \\
[\![\bot]\!]_K &\stackrel{\text{def}}{=} \varnothing \\
[\![p \sqcup q]\!]_K &\stackrel{\text{def}}{=} [\![p]\!]_K \cup [\![q]\!]_K \\
[\![p \otimes q]\!]_K &\stackrel{\text{def}}{=} [\![p]\!]_K \circ [\![q]\!]_K \\
[\![p^+]\!]_K &\stackrel{\text{def}}{=} \bigcup_{n \geq 1} [\![p]\!]_K^n \\
[\![p \to \varphi]\!]_K &\stackrel{\text{def}}{=} \{s \mid \forall \tau\ \mathbf{first}(\tau) = s \text{ and } \tau \in [\![p]\!]_K \\
&\qquad\qquad \Rightarrow \mathbf{last}(\tau) \in [\![\varphi]\!]_K\} \\
[\![\varepsilon]\!]_K &\stackrel{\text{def}}{=} K \\
[\![\Gamma, \Delta]\!]_K &\stackrel{\text{def}}{=} [\![\Gamma]\!]_K \circ [\![\Delta]\!]_K.
\end{aligned}$$

It follows that

$$\begin{aligned}
[\![\bar{b}]\!]_K &= K - [\![b]\!]_K \\
[\![\mathbf{1}]\!]_K &= K \\
[\![p^*]\!]_K &= \bigcup_{n \geq 0} [\![p]\!]_K^n.
\end{aligned}$$

Every trace σ has an associated guarded string $\mathrm{gs}(\sigma)$ defined by

$$\mathrm{gs}(s_0 q_1 s_1 \cdots s_{n-1} q_n s_n) \stackrel{\text{def}}{=} \alpha_0 q_1 \alpha_1 \cdots \alpha_{n-1} q_n \alpha_n,$$

where α_i is the unique atom of B such that $s_i \in [\![\alpha_i]\!]_K$. Thus $\mathrm{gs}(\sigma)$ is the unique guarded string over P, B such that $\sigma \in [\![\mathrm{gs}(\sigma)]\!]_K$.

The sequent $\Gamma \vdash \varphi$ is *valid* in the trace model K if for all traces $\sigma \in [\![\Gamma]\!]_K$, $\mathbf{last}(\sigma) \in [\![\varphi]\!]_K$; equivalently, if $[\![\Gamma]\!]_K \subseteq [\![\Gamma, \varphi]\!]_K$.

The relationship between trace semantics and guarded strings is given by the following lemma.

Lemma 3.2 *In any trace model K, for any program p and trace τ, $\tau \in [\![p]\!]_K$ iff $\mathrm{gs}(\tau) \in GS(p)$. In other words, $[\![p]\!]_K = \mathrm{gs}^{-1}(GS(p))$. The map $X \mapsto \mathrm{gs}^{-1}(X)$ is a* KAT *homomorphism from the algebra of regular sets of guarded strings to the algebra of regular sets of traces over K.*

Proof. Induction on the structure of p. ∎

3.3 Relational Models

Kripke frames (K, m_K) also give rise to relational models. In a relational model, tests, programs, formulas, and environments are interpreted as binary relations on K. Tests and formulas denote subsets of the identity relation.

$$
\begin{aligned}
[p]_K &\stackrel{\text{def}}{=} \mathrm{m}_K(p), \quad p \text{ atomic} \\
[b]_K &\stackrel{\text{def}}{=} \{(s,s) \mid s \in \mathrm{m}_K(b)\}, \quad b \text{ atomic} \\
[\bot]_K &\stackrel{\text{def}}{=} \varnothing \\
[p \sqcup q]_K &\stackrel{\text{def}}{=} [p]_K \cup [q]_K \\
[p \otimes q]_K &\stackrel{\text{def}}{=} [p]_K \circ [q]_K \\
[p^+]_K &\stackrel{\text{def}}{=} \bigcup_{n \geq 1} [p]_K^n \\
[p \to \varphi]_K &\stackrel{\text{def}}{=} \{(s,s) \mid \forall t\, (s,t) \in [p]_K \\
&\qquad\qquad \Rightarrow (t,t) \in [\varphi]_K\} \\
[\varepsilon]_K &\stackrel{\text{def}}{=} \{(s,s) \mid s \in K\} \\
[\Gamma, \Delta]_K &\stackrel{\text{def}}{=} [\Gamma]_K \circ [\Delta]_K.
\end{aligned}
$$

Here $\circ$ denotes ordinary composition of binary relations. It follows that

$$
\begin{aligned}
[\bar{b}]_K &= \{(s,s) \mid (s,s) \notin [b]_K\} \\
[\mathbf{1}]_K &= \{(s,s) \mid s \in K\} \\
[p^*]_K &= \bigcup_{n \geq 0} [p]_K^n.
\end{aligned}
$$

Writing $s \vDash \varphi$ for $(s,s) \in [\varphi]_K$, the defining clause for $p \to \varphi$ becomes

$$s \vDash p \to \varphi \quad \Leftrightarrow \quad \forall t\, (s,t) \in [p]_K \Rightarrow t \vDash \varphi,$$

thus the meaning of $p \to \varphi$ is essentially the same as the meaning of the box formula $[p]\varphi$ of DL.

The sequent $\Gamma \vdash \varphi$ is *valid* in the relational model on (K, m_K) if for all $s, t \in K$, if $(s,t) \in [\Gamma]_K$, then $(t,t) \in [\varphi]_K$; equivalently, if the DL formula $[\Gamma]\varphi$ is true in all states under the rich-test semantics [5], where the environment $\Gamma = \ldots, p, \ldots, \psi, \ldots$ is interpreted as the rich-test program $\cdots; p; \cdots; \psi?; \cdots$.

3.4 Relationship between Trace and Relational Models

It can be shown by induction on syntax that the map

$$r : X \quad \mapsto \quad \{(\mathbf{first}(\sigma), \mathbf{last}(\sigma)) \mid \sigma \in X\}$$

from sets of traces on K to binary relations on K maps $[\![p]\!]_K$ to $[p]_K$ and $[\![\varphi]\!]_K$ to $[\varphi]_K$, using the fact that r commutes with the operators $\cup$ and $\circ$ on sets of traces and binary relations. It follows that validity over relational models is the same as validity over trace models. We include these remarks to establish the connection with the standard relational semantics of DL.

4 A Deductive System

The rules of System S are given in Figure 1. All rules are of the form

$$\frac{\Gamma_1 \vdash \varphi_1 \quad \ldots \quad \Gamma_n \vdash \varphi_n}{\Gamma \vdash \varphi}.$$

The sequents above the line are the *premises* and the sequent below the line is the *conclusion*. Since programs cannot occur positively on the right hand side of $\vdash$, the system has introduction and elimination rules on the left of $\vdash$.

We will use the notation $\Gamma \vdash \varphi$ ambiguously as both an object and a meta-assertion. As an object it denotes a sequent, *i.e.* a sequence of symbols over the appropriate vocabulary. As a meta-assertion it says that the sequent $\Gamma \vdash \varphi$ is provable in S. In particular, $\Gamma \nvdash \varphi$ means that the sequent $\Gamma \vdash \varphi$ is not provable in S. The proper interpretation should always be clear from context.

A rule is *admissible* if for any substitution instance for which the premises are provable, the conclusion is also provable. The proof of the conclusion may depend on the structure of the expressions substituted for the metasymbols appearing in the rule or on the proofs of the premises. To show admissibility, it suffices to derive the conclusion in S augmented with the premises as extra axioms, considering the metasymbols appearing in the rule as atomic symbols in the object language. Any such derivation will then be uniformly valid over all substitution instances.

Axiom: $b \vdash c$, where $b \to c$ is a classical propositional tautology

Arrow Rules:

$$(\mathbf{R}\to)\quad \frac{\Gamma, p \vdash \varphi}{\Gamma \vdash p \to \varphi}$$

$$(\mathbf{I}\to)\quad \frac{\Gamma, p, \psi, \Delta \vdash \varphi}{\Gamma, p \to \psi, p, \Delta \vdash \varphi}$$

Introduction Rules:

$$(\mathbf{I}\otimes)\quad \frac{\Gamma, p, q, \Delta \vdash \varphi}{\Gamma, p \otimes q, \Delta \vdash \varphi}$$

$$(\mathbf{I}\sqcup)\quad \frac{\Gamma, p, \Delta \vdash \varphi \qquad \Gamma, q, \Delta \vdash \varphi}{\Gamma, p \sqcup q, \Delta \vdash \varphi}$$

$$(\mathbf{I}\bot)\quad \Gamma, \bot, \Delta \vdash \varphi$$

$$(\mathbf{I}^+)\quad \frac{q \to \varphi, p \vdash \varphi \qquad q \to \varphi, p, q \vdash \varphi}{q \to \varphi, p^+ \vdash \varphi}$$

Elimination Rules:

$$(\mathbf{E}\otimes)\quad \frac{\Gamma, p \otimes q, \Delta \vdash \varphi}{\Gamma, p, q, \Delta \vdash \varphi}$$

$$(\mathbf{E}^+)\quad \frac{\Gamma, p^+, \Delta \vdash \varphi}{\Gamma, p, \Delta \vdash \varphi}$$

$$(\mathbf{E1}\sqcup)\quad \frac{\Gamma, p \sqcup q, \Delta \vdash \varphi}{\Gamma, p, \Delta \vdash \varphi}$$

$$(\mathbf{E2}\sqcup)\quad \frac{\Gamma, p \sqcup q, \Delta \vdash \varphi}{\Gamma, q, \Delta \vdash \varphi}$$

Structural Rules:

$$(\mathbf{W}\psi)\quad \frac{\Gamma, \Delta \vdash \varphi}{\Gamma, \psi, \Delta \vdash \varphi}$$

$$(\mathbf{W}p)\quad \frac{\Gamma \vdash \varphi}{p, \Gamma \vdash \varphi}$$

$$(\mathbf{CC}^+)\quad \frac{\Gamma, p^+, \Delta \vdash \varphi}{\Gamma, p^+, p^+, \Delta \vdash \varphi}$$

Cut Rule:

$$(\mathbf{cut})\quad \frac{\Gamma \vdash \psi \qquad \Gamma, \psi, \Delta \vdash \varphi}{\Gamma, \Delta \vdash \varphi}$$

Figure 1. Rules of System S

4.1 Basic Properties

Lemma 4.1 *The rule*

$$(\mathbf{E}\,1)\quad \frac{\Gamma, \mathbf{1}, \Delta \vdash \varphi}{\Gamma, \Delta \vdash \varphi}$$

is admissible.

Proof. From $(\mathbf{I}\bot)$ and $(\mathbf{R}\to)$ we get $\Gamma \vdash \mathbf{1}$. The desired conclusion follows from **(cut)**. ∎

Lemma 4.2 *The rule and sequent*

$$(\mathbf{mono})\quad \frac{\varphi \vdash \psi}{p \to \varphi \vdash p \to \psi} \qquad (\mathbf{ident})\quad \varphi \vdash \varphi$$

are admissible.

Proof. The following diagram gives a proof of **(mono)**.

$$\cfrac{\cfrac{\cfrac{\varphi \vdash \psi}{p, \varphi \vdash \psi}\,(\mathbf{W}p)}{p \to \varphi, p \vdash \psi}\,(\mathbf{I}\to)}{p \to \varphi \vdash p \to \psi}\,(\mathbf{R}\to)$$

The identity sequent **(ident)** follows by induction on the structure of φ using **(mono)**. The basis $b \vdash b$ is an instance of the axiom. ∎

Lemma 4.3 *The rules*

$$(\mathbf{MP})\quad \frac{\Gamma \vdash p \to \varphi}{\Gamma, p \vdash \varphi} \qquad (\mathbf{W}\bot)\quad \frac{\Gamma \vdash \bot}{\Gamma, p \vdash \bot}$$

are admissible.

Proof. For **(MP)**, we have $\varphi \vdash \varphi$ by Lemma 4.2. The following figure gives the remainder of the derivation.

$$\cfrac{\Gamma \vdash p \to \varphi \qquad \cfrac{\cfrac{\varphi \vdash \varphi}{p, \varphi \vdash \varphi}\,(\mathbf{W}p)}{\vdots\ (\mathbf{W}p), (\mathbf{W}\psi)}\qquad \cfrac{\Gamma, p, \varphi \vdash \varphi}{\Gamma, p \to \varphi, p \vdash \varphi}\,(\mathbf{I}\to)}{\Gamma, p \vdash \varphi}\,(\mathbf{cut})$$

To derive $(\mathbf{W}\bot)$, the sequent $\Gamma, \bot, p \vdash \bot$ is an instance of $(\mathbf{I}\bot)$. Applying **(cut)** to this and the premise $\Gamma \vdash \bot$ yields the desired conclusion. ∎

We wish to pause and discuss briefly why we view partial correctness reasoning in S as intuitionistic rather than classical. It is not immediately obvious, since formulas are of the form $p_1 \to \cdots \to p_n \to b$, where $p_1, \ldots, p_n$ are programs and b is a test. In particular, formulas are not closed under implication. But we can argue that the implication in

the formula $p \to \varphi$ has intuitionistic flavor by considering the rules that introduce implication. Rule (**R** $\to$) is a typical rule of introduction of implication on the right of $\vdash$. Rule (**I** $\to$) is not so typical, but it can be shown that this rule is derivable from **(ident)**, **(MP)**, (**W** ψ), (**W** p), and **(cut)** as follows.

$$\dfrac{\dfrac{\dfrac{p \to \psi \vdash p \to \psi}{p \to \psi, p \vdash \psi}\ \textbf{(MP)}}{\Gamma, p \to \psi, p \vdash \psi}\ \textbf{(W}\ \psi\textbf{), (W}\ p\textbf{)} \qquad \dfrac{\Gamma, p, \psi, \Delta \vdash \varphi}{\Gamma, p \to \psi, p, \psi, \Delta \vdash \varphi}\ \textbf{(W}\ \psi\textbf{)}}{\Gamma, p \to \psi, p, \Delta \vdash \varphi}\ \textbf{(cut)}$$

Since each of the rules used in the above derivation clearly has an intuitionistic flavor, it follows that (**I** $\to$) has as well.

Lemma 4.4 *The rule*

$$\textbf{(iter)} \quad \frac{\varphi, p \vdash \varphi}{\varphi, p^+ \vdash \varphi}$$

is admissible.

Proof. Taking q in (**I** $^+$) to be **1**, by **(cut)** it suffices to show $\varphi \vdash \mathbf{1} \to \varphi$, $\mathbf{1} \to \varphi \vdash \varphi$, and $\varphi, \mathbf{1}, \mathbf{1} \vdash \varphi$. These follow without difficulty from (**R** $\to$), **(MP)**, **(E 1)**, and (**W** ψ). ∎

Lemma 4.5 *The rules*

$$\textbf{(curry)} \quad \frac{\Gamma, p \to q \to \psi, \Delta \vdash \varphi}{\Gamma, pq \to \psi, \Delta \vdash \varphi}$$

$$\textbf{(uncurry)} \quad \frac{\Gamma, pq \to \psi, \Delta \vdash \varphi}{\Gamma, p \to q \to \psi, \Delta \vdash \varphi}$$

are admissible.

Proof. By **(cut)**, it suffices to show $pq \to \psi \vdash p \to q \to \psi$ and $p \to q \to \psi \vdash pq \to \psi$. For the former, starting with $pq \to \psi \vdash pq \to \psi$, apply **(MP)** and (**E** $\otimes$) to get $pq \to \psi, p, q \vdash \psi$, then apply (**R** $\to$) twice. For the latter, starting with $\psi \vdash \psi$, apply (**W** p) twice to get $p, q, \psi \vdash \psi$, then apply (**I** $\to$) twice to get $p \to q \to \psi, p, q \vdash \psi$. The result then follows from (**I** $\otimes$) and (**R** $\to$). ∎

Lemma 4.6 *Every φ is provably equivalent to some $p \to \bot$ in the sense that $\varphi \vdash p \to \bot$ and $p \to \bot \vdash \varphi$.*

Proof. The formula $q_1 \to \cdots \to q_n \to b$ is equivalent to $q_1 \cdots q_n \overline{b} \to \bot$. The proof of this fact is quite easy using Lemma 4.5 and is left to the reader. ∎

4.2 Relation to Kleene Algebra

We show in this section that S induces a left-handed Kleene algebra structure on programs. Recall that a *Kleene algebra* (KA) is an idempotent semiring such that p^*q is the least solution to $q + px \leq x$ and qp^* is the least solution to $q + xp \leq x$. Equivalently, a Kleene algebra is an idempotent semiring satisfying

$$1 + pp^* = 1 + p^*p = p \tag{1}$$

$$px \leq x \to p^*x \leq x \tag{2}$$

$$xp \leq x \to xp^* \leq x. \tag{3}$$

Boffa [2, 3], based on results of Krob [18], shows that for the equational theory of the regular sets, the right-hand rule (3) is unnecessary. We will call an idempotent semiring satisfying (1) and (2) a *left-handed Kleene algebra*. Boffa's result says that for regular expressions p and q, $R(p) = R(q)$ iff $p = q$ is a logical consequence of the axioms of left-handed Kleene algebra, where R is the usual interpretation of regular expressions as sets of strings.

More specifically, Krob [18] shows that the *classical equations* of Conway [4], along with a certain infinite but independently characterized set of axioms, logically entail all identities of the regular sets over P. The classical equations of Conway are the axioms of idempotent semirings, the equations (1), and the equations

$$\begin{aligned} (p+q)^* &= (p^*q)^*p^* \\ p^* &= p^{**} \\ (pq)^* &= 1 + p(qp)^*q \\ p^* &= (p^n)^*(1+p)^{n-1},\ n \geq 0. \end{aligned}$$

Boffa [2, 3] actually shows that these equations plus the rule

$$p^2 = p \to p^* = 1 + p \tag{4}$$

—which the reader will note is neither left- nor right-handed—imply all the axioms of Krob, therefore the classical equations of Conway plus Boffa's rule (4) are complete for the equational theory of the regular sets over P. The classical equations and Boffa's rule are all easily shown to be theorems of left-handed KA.

Our first task is to extend these results to Kleene algebra with tests and guarded strings.

Lemma 4.7 *Left-handed* KAT *is complete for the equational theory of the regular sets of guarded strings over* P *and* B. *In other words, for every pair of programs p, q in the language of* KAT, *$GS(p) = GS(q)$ if and only if the equation $p = q$ is a logical consequence of the axioms of left-handed* KAT.

Proof. We adapt an argument of [14], in which the same result was proved for KAT with both the left- and right-hand rule. It was shown there that for any program p, there is an equivalent program $\widehat{p}$ such that

(i) $p = \widehat{p}$ is a theorem of KAT, and

(ii) $GS(\widehat{p}) = R(\widehat{p})$, where $R(\widehat{p})$ is the regular set of strings over the alphabet $\mathsf{P} \cup \mathsf{B} \cup \{\overline{b} \mid b \in \mathsf{B}\}$ denoted by $\widehat{p}$ under the usual interpretation of regular expressions.

In other words, any p can be transformed by the axioms of KAT to another program $\widehat{p}$ such that the set of guarded strings denoted by $\widehat{p}$ is the same as the set of strings denoted by $\widehat{p}$.

Now to show completeness of KAT over guarded strings, [14] argued as follows. Suppose $GS(p) = GS(q)$. Then

$$R(\widehat{p}) = GS(\widehat{p}) = GS(p) = GS(q) = GS(\widehat{q}) = R(\widehat{q}).$$

Since KA is complete for the equational theory of the regular sets, $\widehat{p} = \widehat{q}$ is a theorem of KA. Combining this with (i) for p and q implies that $p = q$ is a theorem of KAT.

To adapt this to the present situation, we observe that $\widehat{p} = \widehat{q}$ is a theorem of left-handed KA by the results of Boffa and Krob. Thus in order to complete the proof, we need only ascertain that the right-hand rule (3) is not needed in the proof of $p = \widehat{p}$. This does not follow from Boffa's and Krob's results, since the argument is in KAT, not KA. However, a perusal of [14] reveals that the proof of $p = \widehat{p}$ uses neither the left- or the right-hand rule, but can be carried out using only the classical equations of Conway and the axioms of Boolean algebra. ∎

We now describe the left-handed KAT structure induced by S. Define $p \sqsubseteq q$ if $q \to \varphi \vdash p \to \varphi$ is admissible; that is, if $q \to \varphi \vdash p \to \varphi$ is provable for all φ. Define $p \equiv q$ if $p \sqsubseteq q$ and $q \sqsubseteq p$. The relation $\sqsubseteq$ is a preorder, therefore $\equiv$ is an equivalence relation and $\sqsubseteq$ is a partial order on $\equiv$-classes. Reflexivity is **(ident)** (Lemma 4.2) and transitivity follows from a single application of **(cut)**.

Lemma 4.8 *The operators $\sqcup$ and $\otimes$ are monotone with respect to $\sqsubseteq$. That is, if $p \sqsubseteq q$, then $p \sqcup r \sqsubseteq q \sqcup r$, $pr \sqsubseteq qr$, and $rp \sqsubseteq rq$.*

Proof. The rules **(E1 $\sqcup$)**, **(E2 $\sqcup$)**, and **(I $\sqcup$)** imply that $p \sqcup q$ is the $\sqsubseteq$-least upper bound of p and q modulo $\equiv$. The monotonicity of $\sqcup$ follows by equational reasoning:

$$p \sqsubseteq q \;\Rightarrow\; p \sqsubseteq q \sqcup r \text{ and } r \sqsubseteq q \sqcup r \;\Rightarrow\; p \sqcup r \sqsubseteq q \sqcup r.$$

For $\otimes$, we must show that if $q \to \varphi \vdash p \to \varphi$ for any φ, then $qr \to \varphi \vdash pr \to \varphi$ and $rq \to \varphi \vdash rp \to \varphi$ for any φ. Using **(cut)**, **(curry)**, and **(uncurry)** (Lemma 4.5), it suffices to show that $q \to r \to \varphi \vdash p \to r \to \varphi$ and $r \to q \to \varphi \vdash r \to p \to \varphi$ for any φ. The former is immediate from the assumption, and the latter follows from **(mono)** (Lemma 4.2). ∎

Lemma 4.9 *If $p \sqsubseteq q$ and $qq \sqsubseteq q$, then $p^+ \sqsubseteq q$.*

Proof. Certainly $pq \sqsubseteq q$ by monotonicity. Then

$$\dfrac{\dfrac{\dfrac{q \to \varphi \vdash p \to \varphi}{q \to \varphi, p \vdash \varphi}\,\textbf{(MP)} \quad \dfrac{q \to \varphi \vdash pq \to \varphi}{q \to \varphi, pq \vdash \varphi}\,\textbf{(MP)}}{q \to \varphi, p, q \vdash \varphi}\,\textbf{(E}\otimes\textbf{)}}{q \to \varphi, p^+ \vdash \varphi}\,\textbf{(I}^+\textbf{)}}{q \to \varphi \vdash p^+ \to \varphi}\,\textbf{(R}\to\textbf{)}$$

∎

Lemma 4.10 *Let $\mathcal{P}/\equiv$ denote the set of $\equiv$-equivalence classes. The operations $\sqcup$, $\otimes$, and * are well defined on $\mathcal{P}/\equiv$, and the quotient structure $(\mathcal{P}/\equiv, \sqcup, \otimes, ^*, \bot, \mathbf{1})$ is a left-handed* KA.

Proof. We must argue that all the following properties hold:

$p \sqcup (q \sqcup r) \equiv (p \sqcup q) \sqcup r$	$p(qr) \equiv (pq)r$
$p \sqcup q \equiv q \sqcup p$	$\mathbf{1}p \equiv p\mathbf{1} \equiv p$
$p \sqcup \bot \equiv p$	$\bot p \equiv p\bot \equiv \bot$
$p \sqcup p \equiv p$	$\mathbf{1} \sqcup pp^* \equiv p^*$
$p(q \sqcup r) \equiv pq \sqcup pr$	$\mathbf{1} \sqcup p^*p \equiv p^*$
$(p \sqcup q)r \equiv pr \sqcup qr$	$pq \sqsubseteq q \Rightarrow p^*q \sqsubseteq q.$

These are just the laws of left-handed KA written with the symbols of S.

To derive the distributive law

$$p(q \sqcup r) \quad \sqsubseteq \quad pq \sqcup pr,$$

first from **(MP)**, **(E1 $\sqcup$)**, and **(E $\otimes$)**, one can derive $pq \sqcup pr \to \varphi, p, q \vdash \varphi$ from $pq \sqcup pr \to \varphi \vdash pq \sqcup pr \to \varphi$. Similarly, one can derive $pq \sqcup pr \to \varphi, p, r \vdash \varphi$ using **(E2 $\sqcup$)** instead of **(E1 $\sqcup$)**. Then

$$\dfrac{\dfrac{pq \sqcup pr \to \varphi, p, q \vdash \varphi \qquad pq \sqcup pr \to \varphi, p, r \vdash \varphi}{pq \sqcup pr \to \varphi, p, q \sqcup r \vdash \varphi}\,\textbf{(I}\sqcup\textbf{)}}{pq \sqcup pr \to \varphi \vdash p(q \sqcup r) \to \varphi}\,\textbf{(I}\otimes\textbf{), (R}\to\textbf{)}$$

All the other axioms of idempotent semirings follow in an equally straightforward manner. Since $\sqcup$ and $\otimes$ are monotone with respect to $\sqsubseteq$ (Lemma 4.8), they are well defined on $\equiv$-classes.

The inequality $p^+p^+ \sqsubseteq p^+$ follows from **(CC $^+$)** by:

$$\dfrac{\dfrac{\dfrac{p^+ \to \varphi \vdash p^+ \to \varphi}{p^+ \to \varphi, p^+ \vdash \varphi}\,\textbf{(MP)}}{p^+ \to \varphi, p^+, p^+ \vdash \varphi}\,\textbf{(CC}^+\textbf{)}}{p^+ \to \varphi \vdash p^+p^+ \to \varphi}\,\textbf{(I}\otimes\textbf{), (R}\to\textbf{)}$$

The inequality $p \sqsubseteq p^+$ follows from $(\mathbf{E}^+)$ in a similar fashion. Monotonicity of $^+$ and * then follow from Lemma 4.9 by equational reasoning:

$$\begin{aligned} p \sqsubseteq q &\Rightarrow p \sqsubseteq q^+ \text{ and } q^+q^+ \sqsubseteq q^+ \\ &\Rightarrow p^+ \sqsubseteq q^+ \end{aligned}$$

$$p \sqsubseteq q \Rightarrow p^* = \mathbf{1} \sqcup p^+ \sqsubseteq \mathbf{1} \sqcup q^+ = q^*.$$

We now prove the KA identities involving *. Arguing equationally, we have

$$p \sqcup pp^+ \sqsubseteq p^+ \sqcup p^+p^+ \sqsubseteq p^+ \sqcup p^+ \sqsubseteq p^+,$$

and similarly $p \sqcup p^+p \sqsubseteq p^+$. For the opposite inequalities we will use Lemma 4.9. Clearly we have $p \sqsubseteq p \sqcup pp^+$. We also have $pp \sqsubseteq pp^+$, $ppp^+ \sqsubseteq pp^+$, $pp^+p \sqsubseteq pp^+$ and $pp^+pp^+ \sqsubseteq pp^+$, hence

$$(p \sqcup pp^+)(p \sqcup pp^+) \sqsubseteq pp^+ \sqsubseteq p \sqcup pp^+.$$

By Lemma 4.9, $p^+ \sqsubseteq p \sqcup pp^+$. Since the opposite inequality was already established, we have $p^+ \equiv p \sqcup pp^+$.

Now we can show that $\mathbf{1} \sqcup pp^* \equiv p^*$:

$$\begin{aligned} p^* &\equiv \mathbf{1} \sqcup p^+ \equiv \mathbf{1} \sqcup p \sqcup pp^+ \equiv \mathbf{1} \sqcup p(\mathbf{1} \sqcup p^+) \\ &\equiv \mathbf{1} \sqcup pp^*. \end{aligned}$$

The identities $p^+ \equiv p \sqcup p^+p$ and $\mathbf{1} \sqcup p^*p \equiv p^*$ are obtained in a similar fashion.

It remains to show $pq \sqsubseteq q \Rightarrow p^*q \sqsubseteq q$. This is established by the following derivation:

$$\dfrac{\dfrac{q \to \varphi \vdash q \to \varphi}{q \to \varphi, \mathbf{1} \vdash q \to \varphi}\ (\mathbf{W}\,\psi) \qquad \dfrac{\dfrac{\dfrac{\dfrac{q \to \varphi \vdash pq \to \varphi}{q \to \varphi, pq \vdash \varphi}\ (\mathbf{E}\otimes)}{q \to \varphi, p, q \vdash \varphi}\ (\mathbf{R}\to)}{q \to \varphi, p \vdash q \to \varphi}\ (\text{iter})}{q \to \varphi, p^+ \vdash q \to \varphi}\ (\mathbf{I}\sqcup)}{\dfrac{q \to \varphi, \mathbf{1} \sqcup p^+ \vdash q \to \varphi}{q \to \varphi \vdash (\mathbf{1} \sqcup p^+)q \to \varphi}\ (\mathbf{MP}).\ (\mathbf{I}\otimes).\ (\mathbf{R}\to)}$$

∎

Lemma 4.11 *If $b \to c$ is a classical tautology, then $b \sqsubseteq c$. Thus the tests form a Boolean algebra modulo $\equiv$.*

Proof. We have $c \to \varphi, b \vdash c$ by the axiom $b \vdash c$ and the weakening rule $(\mathbf{W}\,\psi)$, and we have $c \to \varphi, c \vdash \varphi$ by **(MP)**. The desired conclusion $c \to \varphi \vdash b \to \varphi$ then follows from **(cut)** and $(\mathbf{R}\to)$. ∎

Combining Lemmas 4.10 and 4.11 and the fact that the regular sets of guarded strings form the free KAT on generators P and B, we have

Lemma 4.12 *The structure $(\mathcal{P}/\equiv, \mathcal{B}/\equiv, \sqcup, \otimes, ^*, \bar{}, \bot, \mathbf{1})$ is a left-handed* KAT *and is isomorphic to the algebra of regular sets of guarded strings over* P *and* B*. Thus for any programs p and q, $p \sqsubseteq q$ iff $GS(p) \subseteq GS(q)$ and $p \equiv q$ iff $GS(p) = GS(q)$.*

5 Soundness

Theorem 5.1 *If $\Gamma \vdash \varphi$ is provable, then it is valid in all trace and relational models.*

Proof. We need only show soundness over trace models. This is easily established by induction on proofs in S with one case for each proof rule. We argue the cases **(cut)** and $(\mathbf{I}\to)$ explicitly.

For **(cut)**, we need to show that

$$[\![\Gamma, \Delta]\!]_K \subseteq [\![\Gamma, \Delta, \varphi]\!]_K$$

under the assumptions

$$\begin{aligned} [\![\Gamma]\!]_K &\subseteq [\![\Gamma, \psi]\!]_K \\ [\![\Gamma, \psi, \Delta]\!]_K &\subseteq [\![\Gamma, \psi, \Delta, \varphi]\!]_K. \end{aligned}$$

Using monotonicity of $\circ$,

$$\begin{aligned} &[\![\Gamma, \Delta]\!]_K \\ &\quad = [\![\Gamma]\!]_K \circ [\![\Delta]\!]_K \\ &\quad \subseteq [\![\Gamma, \psi]\!]_K \circ [\![\Delta]\!]_K \\ &\quad = [\![\Gamma, \psi, \Delta]\!]_K \\ &\quad \subseteq [\![\Gamma, \psi, \Delta, \varphi]\!]_K \\ &\quad = [\![\Gamma]\!]_K \circ [\![\psi]\!]_K \circ [\![\Delta, \varphi]\!]_K \\ &\quad \subseteq [\![\Gamma]\!]_K \circ [\![\mathbf{1}]\!]_K \circ [\![\Delta, \varphi]\!]_K \\ &\quad = [\![\Gamma]\!]_K \circ [\![\Delta, \varphi]\!]_K \\ &\quad = [\![\Gamma, \Delta, \varphi]\!]_K. \end{aligned}$$

For $(\mathbf{I}\to)$, we want to show that if

$$[\![\Gamma, p, \psi, \Delta]\!]_K \subseteq \mathbf{last}^{-1}([\![\varphi]\!]_K),$$

then

$$[\![\Gamma, p \to \psi, p, \Delta]\!]_K \subseteq \mathbf{last}^{-1}([\![\varphi]\!]_K).$$

It suffices to show that

$$[\![p \to \psi]\!]_K \circ [\![p]\!]_K \subseteq [\![p]\!]_K \circ [\![\psi]\!]_K.$$

But

$$\begin{aligned} &\tau \in [\![p \to \psi]\!]_K \circ [\![p]\!]_K \\ &\quad \Leftrightarrow \mathbf{first}(\tau) \in [\![p \to \psi]\!]_K \text{ and } \tau \in [\![p]\!]_K \\ &\quad \Rightarrow \tau \in [\![p]\!]_K \text{ and } \mathbf{last}(\tau) \in [\![\psi]\!]_K \\ &\quad \Leftrightarrow \tau \in [\![p]\!]_K \circ [\![\psi]\!]_K. \end{aligned}$$

The other cases are equally straightforward. ∎

6 Completeness

Theorem 6.1 *If $\Gamma \not\vdash \varphi$, then there exist an acyclic trace model K and a trace $\sigma \in [\![\Gamma]\!]_K$ such that* $\mathbf{last}(\sigma) \notin [\![\varphi]\!]_K$.

Proof. By Lemma 4.6, we can assume without loss of generality that φ is of the form $p \to \bot$. The proof proceeds by induction on the length of Γ. For the basis of the induction, suppose Γ is empty, so that $\not\vdash p \to \bot$. Then $p \not\equiv \bot$. By Lemma 4.12, $GS(p) \neq \varnothing$. Construct a Kripke frame K consisting of a single acyclic trace σ such that $\mathrm{gs}(\sigma) \in GS(p)$. By Lemma 3.2, $\sigma \in [\![p]\!]_K$. Then $\mathbf{first}(\sigma) \in [\![\varepsilon]\!]_K$ and $\mathbf{first}(\sigma) \notin [\![p \to \bot]\!]_K$.

For the induction step in which the environment ends with a program, say $\Gamma, p \not\vdash \varphi$, we have $\Gamma \not\vdash p \to \varphi$ by **(MP)**. Applying the induction hypothesis, there exist an acyclic trace model K and traces σ and τ such that $\sigma \in [\![\Gamma]\!]_K$, $\mathbf{last}(\sigma) = \mathbf{first}(\tau)$, $\tau \in [\![p]\!]_K$, and $\mathbf{last}(\tau) \notin [\![\varphi]\!]_K$. Then $\sigma\tau \in [\![\Gamma, p]\!]_K$ and $\mathbf{last}(\sigma\tau) \notin [\![\varphi]\!]_K$.

Finally, we argue the induction step in which the environment ends with a formula, say $\Gamma, \psi \not\vdash \varphi$. By Lemma 4.6, we can rewrite this as $\Gamma, q \to \bot \not\vdash p \to \bot$. Let w be an expression representing the set of all guarded strings (see Lemma 3.1). Let r and s be programs such that $GS(r) = GS(p) \cap GS(qw)$ and $GS(s) = GS(p) - GS(qw)$. These programs exist by Lemma 3.1, and $GS(p) = GS(r \sqcup s)$. By Lemma 4.12, we can replace p by $r \sqcup s$ to get $\Gamma, q \to \bot \not\vdash r \sqcup s \to \bot$. By **(R $\to$)**, $\Gamma, q \to \bot, r \sqcup s \not\vdash \bot$, and by **(I $\sqcup$)**, either $\Gamma, q \to \bot, r \not\vdash \bot$ or $\Gamma, q \to \bot, s \not\vdash \bot$. But it cannot be the former, since $\Gamma, q \to \bot, q, w \vdash \bot$, therefore $\Gamma, q \to \bot \vdash qw \to \bot$, and by Lemma 4.12, $r \sqsubseteq qw$, therefore by **(cut)**, $\Gamma, q \to \bot \vdash r \to \bot$.

Thus it must be the case that $\Gamma, q \to \bot, s \not\vdash \bot$, so $\Gamma, q \to \bot \not\vdash s \to \bot$. By weakening we have $\Gamma \not\vdash s \to \bot$. Then by the induction hypothesis, there exist an acyclic trace model K and traces $\sigma \in [\![\Gamma]\!]_K$ and $\tau \in [\![s]\!]_K$ such that $\mathbf{last}(\sigma) = \mathbf{first}(\tau)$. Construct a trace model M consisting only of the acyclic trace $\sigma\tau$. By Lemma 3.2, $\tau \notin [\![qw]\!]_M$, therefore no prefix of τ is in $[\![q]\!]_M$. Then $\mathbf{last}(\sigma) \in [\![q \to \bot]\!]_M$, therefore $\sigma \in [\![\Gamma, q \to \bot]\!]_M$. Moreover, $\mathbf{last}(\sigma) \notin [\![p \to \bot]\!]_M$, since $\mathbf{last}(\sigma) = \mathbf{first}(\tau)$ and $\tau \in [\![p]\!]_M$. ∎

7 Conclusions and Future Work

It has recently been shown that deciding whether a given sequent is valid is *PSPACE*-complete [12]. Several interesting questions present themselves for further investigation.

1. The completeness proof relies on the results of Boffa [2, 3], which are based in turn on the results of Krob [18]. Krob's proof is fairly involved, comprising an entire journal issue. One would like to have a proof of completeness based on first principles.

2. The relative expressive and deductive power of S compared with similar systems such as KAT, PDL, and PHL is not completely understood. S is at least as expressive as PHL and the equational theory of KAT, and apparently more so, since it is not clear how to express general sequents $\varphi_1, p_1, \varphi_2, \ldots, p_{n-1}, \varphi_n \vdash \psi$ in PHL or KAT. On the other hand, it is not clear how to express general Horn formulas of KA such as $px = xq \to p^*x = xq^*$ in S.

3. Application of the linear implication operator $\to$ is limited to programs on the left-hand side and formulas on the right-hand side. It would be interesting to see whether more general forms correspond to anything useful and whether the system can be extended to handle them. The operator $\to$ is a form of residuation (see [19, 10]), and this connection bears further investigation.

4. We would like to extend S to handle liveness properties and total correctness.

5. We would like to undertake a deeper investigation into the structure of proofs with an eye toward establishing normal form and cut elimination theorems.

Acknowledgements

We thank Riccardo Pucella for pointing out an error in an earlier draft and the anonymous reviewers for their valuable comments. The support of the National Science Foundation under grant CCR-9708915 and Polish KBN Grant 7 T11C 028 20 is gratefully acknowledged.

References

[1] S. Artemov. Explicit provability and constructive semantics. *Bull. Symbolic Logic*, 7(1):1–36, March 2001.

[2] M. Boffa. Une remarque sur les systèmes complets d'identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications*, 24(4):419–423, 1990.

[3] M. Boffa. Une condition impliquant toutes les identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications*, 29(6):515–518, 1995.

[4] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[5] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[8] K. Gödel. Eine Interpretation des intuitionistischen Aussagenkalküls. *Ergebnisse eines mathematischen Kolloquiums*, 4:39–40, 1933. Reprinted in: S. Feferman, ed., Collected Works of Kurt Gödel, v. 1, New York, Oxford University Press, 1986.

[9] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, 2000.

[10] D. Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994.

[11] D. Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.

[12] D. Kozen. Automata on guarded strings and applications. Technical Report 2001-1833, Computer Science Department, Cornell University, January 2001.

[13] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Proc. 1st Int. Conf. Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582, London, July 2000. Springer-Verlag.

[14] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

[15] D. Kozen and J. Tiuryn. On the completeness of propositional Hoare logic. In J. Desharnais, editor, *Proc. 5th Int. Seminar Relational Methods in Computer Science (RelMiCS 2000)*, pages 195–202, January 2000.

[16] S. Kripke. Semantic analysis of modal logic. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 9:67–96, 1963.

[17] S. Kripke. Semantical analysis of intuitionistic logic I. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, 1965.

[18] D. Krob. A complete system of B-rational identities. *Theoretical Computer Science*, 89(2):207–343, October 1991.

[19] V. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Proc. Logics in AI: European Workshop JELIA '90*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120, New York, September 1990. Springer-Verlag.

[20] A. S. Troelstra. *Lectures on Linear Logic*, volume 29 of *CSLI Lecture Notes*. Center for the Study of Language and Information, 1992.

[21] D. N. Yetter. Quantales and (noncommutative) linear logic. *J. Symbolic Logic*, 55:41–64, 1990.

Perturbed Turing Machines and Hybrid Systems

Eugene Asarin *
VERIMAG,
2 av. de Vignate, 38610 Gières, France.
Email: Eugene.Asarin@imag.fr
Phone: (+33) 4 76 63 48 33, Fax: (+33) 4 76 63 48 50.

Ahmed Bouajjani
LIAFA, University of Paris 7, Case 7014,
2 Place Jussieu, 75251 Paris cedex 5, France.
Email: abou@liafa.jussieu.fr
Phone: (+33) 1 44 27 78 19, Fax: (+33) 1 44 27 68 49.

Abstract

We investigate the computational power of several models of dynamical systems under infinitesimal perturbations of their dynamics. We consider in our study models for discrete and continuous time dynamical systems: Turing machines, Piecewise affine maps, Linear hybrid automata and Piecewise constant derivative systems (a simple model of hybrid systems). We associate with each of these models a notion of perturbed dynamics by a small ε (w.r.t. to a suitable metrics), and define the perturbed reachability relation *as the intersection of all reachability relations obtained by ε-perturbations, for all possible values of ε. We show that for the four kinds of models we consider, the perturbed reachability relation is co-recursively enumerable, and that any co-r.e. relation can be defined as the perturbed reachability relation of such models. A corollary of this result is that systems that are robust, i.e., their reachability relation is stable under infinitesimal perturbation, are decidable.*

1 Introduction

Recently, the investigation of the relations between dynamics and computation attracted attention of several research communities (see e.g. [1] where Turing machines are considered as dynamical systems, and [2] and [3] where discrete and continuous time dynamical systems are considered as computation models).

Our initial motivation for this research was related to hybrid systems (see e.g. [4]). Since the first undecidability results were stated for hybrid systems (such as Linear hybrid automata [5] or Piecewise constant derivative systems [3]), a folklore conjecture appeared, saying that this undecidability is due to non-stability, non-robustness, sensitivity to initial values of the systems, and that it never occurs in "real" systems. There were several attempts to formalize and to prove (or to disprove) this conjecture [6, 7] (cf. *Related Work* below). We think however that this conjecture is more rich than these formalizations and that exploring relations between complexity of behaviours of a dynamical system (not necessarily hybrid) and its properties related to stability, robustness, chaos is an important scientific challenge (see [8]).

In this paper we explore one facet of this problem: how small perturbations of dynamics influence the computational power of the system. We consider different kinds of transition systems corresponding to widely used models of dynamical systems: Turing machines (TM), Piecewise affine maps (PAM), Linear hybrid automata (LHA), and Piecewise constant derivative (PCD) systems. We introduce for these models a notion of "perturbed" dynamics and study the computational power of the corresponding perturbed systems. Perturbations are defined for each model using a notion of metrics on the state space (allowing to define how distant is the ideal dynamics from the perturbed one). The notion of *small* perturbation is easier to understand for computational models with a continuous state-space (such that PCD, LHA, and PAM) than for discrete ones like TM. For such models, given a transition system with a reachability relation R, the idea is to perturb the dynamics by a small ε, and then, to take (as the perturbed dynamics of the system) the limit (intersection) R_ω of the perturbed reachability relations as this ε tends to 0. We say that a system is *robust* if its reachability relation does not change under small perturbations of the dynamics, i.e., R is equal to R_ω.

We show that for the three models of PAM, LHA, and PCD, the relation R_ω belongs to the class Π^0_1 (i.e. it is co-recursively enumerable), and moreover, any Π^0_1 relation can be reduced to a relation R_ω of a perturbed system. In other words, any complement to a r.e. set can be semi-decided by an infinitesimally perturbed system. This result is somehow surprising since it means that noise by itself does not make the reachability problem decidable, but it transforms it in a rather non-trivial way (from Σ^0_1 to Π^0_1). Furthermore, an immediate corollary of the result above is the following fact: the reachability problem is decidable for the class of robust systems.

*The work of this author was supported in part by the NATO under grant CRG-961115

0-7695-1281-X/01 $10.00 © 2001 IEEE

In the case of Turing machines, the analogous notion of small perturbation is obtained by considering the prefix distance (Cantor distance) as metrics on the set of tape configurations. In fact, this metrics is an adequate characteristics for these machines; in particular, the dynamics of these machines has good properties w.r.t. this metrics, e.g., the transition function of a TM is always Lipshitz w.r.t. it (see [1] for a detailed argument). So, we consider that a TM is subjected to a small noise if its configuration is slightly perturbed in the sense of this metrics, or equivalently, all the perturbations of the tape content happen far from the head. Similarly to the other models, given a TM recognizing a language L, for every natural number n, we define L_n to be the set of all words that are accepted if we allow perturbations (arbitrary changes in the tape) beyond a distance n from the head, and we take L_ω to be the intersection of all the languages L_n. It can be understood intuitively that the notion of robustness of a TM according to this notion of perturbation actually coincides with the notion of boundedness since only machines that can visit arbitrarily far positions from their initial position can have a different perturbed language. We prove that for TM also the same results as for the other models hold: the language L_ω is in Π_1^0, and every Π_1^0 language can be represented as a perturbed language of a TM, which means that robust TM's correspond precisely to machines recognizing recursive languages.

We give in the paper the proofs for the models mentioned above in an increasing technical complexity order. The TM case unveils the mechanism of the effect of perturbation and allows to understand the essence of this mechanism on a common and relatively simple model. The PAM case makes it clear how this mechanism works in the continuous state space, without unneeded technical complexity. Essentially the same techniques used for PAM can also be applied to the more popular model of LHA (we omit in this extended abstract the proofs concerning LHA). Moreover, the proof for PAM is a good introduction to the trickier one for PCD, which is a simple and natural model for hybrid systems, and perhaps the most motivating case.

Related work. Recently, a similar approach to ours was independently invented and applied in a completely different context to the analysis of numerical methods for chaotic dynamical systems by Kloeden and Kozyakin. In [9], they refer to the procedure of infinitesimal perturbation of dynamical systems as *inflation*.

The notion of perturbation we use (especially in the case of continuous state space systems) was inspired by the work of Anuj Puri who studied the reachability relation of timed automata (with finitely many control states) under infinitesimal perturbation [10]. He showed that for these models, the perturbed reachability relation is still decidable and he gives an effective representation of this relation. Our work concerns models that are more general than timed automata, and aims to show that infinitesimal perturbation has the same effect on several common models of dynamical systems, namely that the perturbed dynamics corresponds in all cases to a co-recursively enumerable relation (set), and that robustness coincides with decidability.

Concerning the decidability issue of the reachability problem, there are two works closely related to ours [6, 7]: Martin Fränzle has shown in [6] a similar result to ours for a certain model of hybrid systems. Our work shows that the fact that "*robustness implies decidability*" can be proved for other different types of transitions systems. Moreover, our hardness results (inverse implication) show that the relation between robustness and decidability is really tight. Our result is in contrast with Thomas Henzinger's result [7] stating that reachability is still undecidable for hybrid systems that allow small perturbations *of the trajectory*. It is interesting to see that a small semantical difference between these two approaches drastically changes the complexity.

Finally, the effect of noise on the power of analog computational models and the dependence of this power from the level of this noise are explored in [11, 12, 13]. Differently, we consider in our work the limit behavior with noise level tending to zero.

Outline. The rest of the paper is organized as follows: in section 2 we define the computation models (kinds of dynamical systems) we consider: TM, PAM, and PCD, and their perturbed versions. In sections 3–5 we formulate and prove the main results for these models. For lack of space, we omit here the case of LHA since the proofs concerning these models are technically very similar to those for PAM.

Acknowledgments. We would like to thank Vincent Blondel, Victor Kozyakin, Oded Maler and Anuj Puri for useful discussion.

2 Perturbed Models

2.1 Perturbed Turing machines (PTMs)

Let us recall the definition of a *Turing machine* (TM for short) (see figure 1(a)).

Let Σ be a finite alphabet, and let B be a special symbol $B \notin \Sigma$. A TM over Σ is a tuple (Q, q_{init}, F, Γ) where Q is a finite set of control states, $q_{init} \in Q$ is the initial control state, $F \subseteq Q$ is a set of *accepting* states, and Γ is a set of transitions of the form $(q, a) \to (q', b, \delta)$ where $q, q' \in Q$, $a, b \in \Sigma \cup \{B\}$, and $\delta \in \{-1, 0, 1\}$.

A configuration of the machine is an unbounded sequence (from left and right) of the form

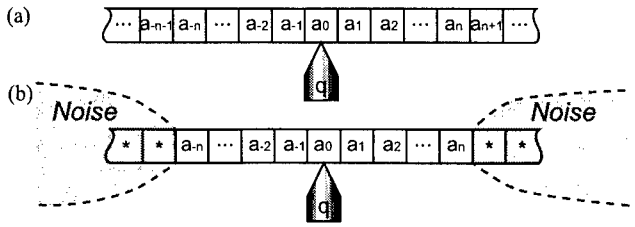


Figure 1: (a) A Turing machine. (b) Its n-perturbed version.

$\cdots a_{-2}a_{-1}[q, a_0]a_1a_2\cdots$ where the a_is are symbols in $\Sigma \cup \{B\}$. Intuitively, $[q, a_0]$ means that the current control state of is q and that the head of the machine is at symbol a_0.

Given a transition $(q, a) \rightarrow (q', b, \delta)$ in Γ, if the symbol pointed by the head of the machine is equal to a, then the machine can change its configuration in the following manner: the symbol pointed by the head is replaced by b and then the head is moved to the left or to the right, or it stays at the same position according to whether δ is -1, 1, or 0, respectively.

Let $w = a_1, \cdots, a_n$ be a word in Σ^*. We say that w is accepted by $\mathcal{M}$ if, starting from the configuration $\cdots BBB[q_{init}, a_1]\cdots a_nBBB\cdots$ the machine $\mathcal{M}$ eventually stops in an accepting state. Let $L(\mathcal{M})$ denote the set of such words, i.e., the recursively enumerable (r.e.) language semi-recognized by $\mathcal{M}$.

Now, let us introduce the concept of *perturbed Turing machines* (PTMs for short). Given an integer $n > 0$, the n-perturbed version of the machine $\mathcal{M}$ is defined exactly as $\mathcal{M}$ except that before any transition all the symbols at the distance n or more from the head of the machine can be altered (i.e., replaced by other symbols) arbitrarily: Given a configuration

$$\cdots a_{-n-1}a_{-n}a_{-n+1}\cdots a_{-1}[q, a_0]a_1\cdots a_{n-1}a_na_{n+1}\cdots$$

the n-perturbed version of $\mathcal{M}$ may replace any symbols to the left of a_{-n} (starting from a_{-n-1}) and to the right of a_n (starting from a_{n+1}) by any other symbols in $\Sigma \cup \{B\}$ before executing a transition of $\mathcal{M}$ (at a_0). Hence, the machine becomes a nondeterministic transition system (see figure 1(b)).

A word w is accepted by the n-perturbed version of $\mathcal{M}$ if there exists a run of this machine which stops in an accepting state. Let $L_n(\mathcal{M})$ be the n-perturbed language of $\mathcal{M}$, i.e., the set of words in Σ^* that are accepted by the n-perturbed version of $\mathcal{M}$.

It is easy to see that if a word is accepted by $\mathcal{M}$, then it can also be recognized by all the n-perturbed versions of $\mathcal{M}$, for every $n > 0$ (perturbed machines have more behaviors). Moreover, if the $(n+1)$-perturbed version accepts a word w, the n-perturbed version will also accept it since obviously all alterations at distance greater than $n+1$ from the head can also happen in the n-perturbed machine. Hence, we have:

Lemma 1 $L_1(\mathcal{M}) \supseteq L_2(\mathcal{M}) \supseteq \cdots \supseteq L(\mathcal{M})$

This technically justifies the following crucial definition (explained in the introduction): *ω-perturbed language* of the machine $\mathcal{M}$ is given by

$$L_\omega(\mathcal{M}) = \bigcap_n L_n(\mathcal{M})$$

Informally speaking, $L_\omega(\mathcal{M})$ consists of all the words that can be accepted by $\mathcal{M}$ when it is subject to arbitrarily "small" perturbations. The previous lemma could be trivially extended to:

Lemma 2 $L_1(\mathcal{M}) \supseteq L_2(\mathcal{M}) \supseteq \cdots \supseteq L_\omega(\mathcal{M}) \supseteq L(\mathcal{M})$

2.2 Piecewise affine maps

The second kind of systems to which we apply small perturbations was introduced as a computation model in [2]. Recall some definitions and results from that paper.

Definition 1 (PAM System) *A Piecewise affine map system (PAM) is a discrete-time dynamical system $\mathcal{P}$ defined by an assignment $\mathbf{x} := f(\mathbf{x})$ on a bounded polyhedral set $X \subset \mathbb{R}^d$, where f is a (possibly partial) function from X to X represented by a formula:*

$$f(\mathbf{x}) = A_i\mathbf{x} + \mathbf{b}_i \text{ for } \mathbf{x} \in P_i, \quad i = 1..N$$

where A_i are rational $d \times d$-matrices, $\mathbf{b}_i \in \mathbb{Q}^d$ and P_i are rational polyhedral sets in X.

A *trajectory* of $\mathcal{P}$ is a sequence $\mathbf{x}_n$ evolving according to f, i.e. such that $\mathbf{x}_{n+1} = f(\mathbf{x}_n)$ for all n.

In other words, a PAM system consists of partitioning the space into convex polyhedral sets ("regions"), and assigning an affine update rule $\mathbf{x} := A_i\mathbf{x} + \mathbf{b}_i$ to all the points sharing the same region (see figure 2 (a)).

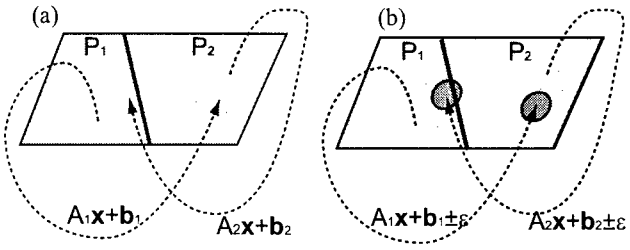


Figure 2: (a) A 2-dimensional PAM system with 2 regions. (b) Its ε-perturbed version.

It is important to emphasize that since we assume that all constants in the system's definition are *rational*, the expressive power of PAM is *not* achieved using the introduction of some non-computable real numbers.

To each PAM $\mathcal{P}$ we associate its reachability relation $R^{\mathcal{P}}(\cdot,\cdot)$ on $\mathcal{Q}^d$. Namely, for two rational points $\mathbf{x}$ and $\mathbf{y}$ the relation $R^{\mathcal{P}}(\mathbf{x},\mathbf{y})$ holds iff there exists a trajectory of $\mathcal{P}$ from $\mathbf{x}$ to $\mathbf{y}$.

The following result on the computational power of PAMs was proved in [14, 2]

Theorem 1 (Simulation of TM by PAM) *Let $\mathcal{M}$ be a TM. We can effectively construct a PAM $\mathcal{P}$ and an encoding $e : \Sigma^* \to \mathcal{Q}^d$ such that for any word w the following equivalence holds. $w \in L(\mathcal{M})$ iff $R^{\mathcal{P}}(e(w), O)$, where O denotes the origin in $\mathbb{R}^d$.*

The following characterization of the complexity of the reachability relation is now immediate:

Corollary 1 (Computational power of PAM)

- *For any PAM $\mathcal{P}$ its reachability relation is r.e.*
- *Any r.e. set S is 1-reducible (see [15]) to the reachability relation of a PAM.*

2.3 Perturbed PAMs (PPAMs)

Now we can apply the paradigm of small perturbations to PAMs. Consider a PAM $\mathcal{P}$ described by the assignment $\mathbf{x} := f(x)$. For any $\varepsilon > 0$ we consider the ε-perturbed system $\mathcal{P}_\varepsilon$ (see figure 2 (b)). Its trajectories are defined as sequences $\mathbf{x}_n$ satisfying the inequality $||\mathbf{x}_{n+1} - f(\mathbf{x}_n)|| < \varepsilon$ for all n. This non-deterministic system can be considered as $\mathcal{P}$ submitted to a small noise with magnitude ε. We denote reachability in the system $\mathcal{P}_\varepsilon$ by $R^{\mathcal{P}}_\varepsilon(\cdot,\cdot)$. All trajectories of a non-perturbed system $\mathcal{P}$ are also trajectories of the ε-perturbed system $\mathcal{P}_\varepsilon$. If $\varepsilon_1 < \varepsilon_2$ then any trajectory of the ε_1-perturbed system is also a trajectory of the ε_2-perturbed PAM.

Like for TM we can pass to a limit for $\varepsilon \to 0$. Namely $R^{\mathcal{P}}_\omega(\mathbf{x},\mathbf{y})$ iff $\forall \varepsilon > 0 \quad R^{\mathcal{P}}_\varepsilon(\mathbf{x},\mathbf{y})$. This means reachability with arbitrarily small perturbing noise.

The following analog of Lemmata 1 and 2 is now immediate:

Lemma 3 *For any $\varepsilon_2 > \varepsilon_1 > 0$ and rational points $\mathbf{x}$ and $\mathbf{y}$ the following implications hold: $R^{\mathcal{P}}(\mathbf{x},\mathbf{y}) \Rightarrow R^{\mathcal{P}}_\omega(\mathbf{x},\mathbf{y}) \Rightarrow R^{\mathcal{P}}_{\varepsilon_1}(\mathbf{x},\mathbf{y}) \Rightarrow R^{\mathcal{P}}_{\varepsilon_2}(\mathbf{x},\mathbf{y})$*

2.4 Piecewise Constant Derivative Hybrid Systems (PCDs)

The last kind of systems to which we apply small perturbations was introduced in [3] in the context of hybrid systems. Recall some definitions and results.

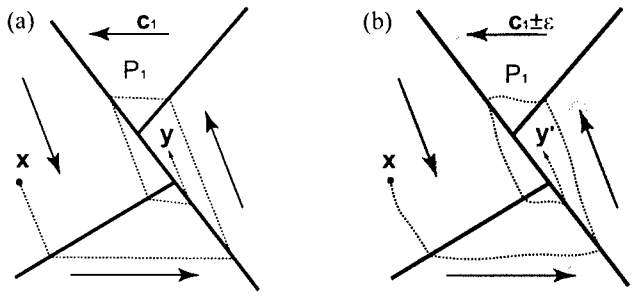


Figure 3: (a) A 2-dimensional PCD system with 4 regions and a trajectory from $\mathbf{x}$ to $\mathbf{y}$. (b) The ε-perturbed version of this PCD.

Definition 2 (PCD System) *A piecewise-constant derivative (PCD) system is a continuous-time dynamical system $\mathcal{H}$ defined by a differential equation $\dot{\mathbf{x}} = f(\mathbf{x})$ on a bounded polyhedral set $X \subset \mathbb{R}^d$ (the state-space), where f is a (possibly partial) function from X to $\mathbb{R}^d$ represented by a formula:*

$$f(\mathbf{x}) = \mathbf{c}_i \text{ for } \mathbf{x} \in P_i. \quad i = 1..N$$

where $\mathbf{c}_i \in \mathcal{Q}^d$ and P_i are rational polyhedral sets in X.

A *trajectory* of $\mathcal{H}$ starting at some $\mathbf{x}_0 \in X$ is a solution of the differential equation with initial condition $\mathbf{x} = \mathbf{x}_0$, defined as a continuous function $\xi : \mathbb{R}^+ \to X$ such that $\xi(0) = \mathbf{x}_0$ and for every t, $f(\xi(t))$ is defined and is equal to the *right derivative* of $\xi(t)$.

In other words, a PCD system consists of partitioning the space into convex polyhedral sets ("regions"), and assigning a constant derivative $\mathbf{c}$ ("slope") to all the points sharing the same region (see figure 3 (a)). The trajectories of such systems are broken lines, with the breakpoints occurring on the boundaries of the regions. In order to rule out some pathologies we consider only PCDs $\mathcal{H}$ which satisfy an additional assumption of being *strongly non-zeno* i.e. the time interval between two consecutive visits of the same region should be bounded from below by a positive constant Δ.

To each PCD $\mathcal{H}$ we associate its reachability relation $R^{\mathcal{H}}(\cdot,\cdot)$ on $\mathcal{Q}^d$. Namely, for two rational points $\mathbf{x}$ and $\mathbf{y}$ the relation $R^{\mathcal{H}}(\mathbf{x},\mathbf{y})$ holds iff there exists a trajectory of $\mathcal{H}$ from $\mathbf{x}$ to $\mathbf{y}$.

The following result on the computational power of PCDs was proved in [3]

Theorem 2 (Simulation of TM by PCD) *Let $\mathcal{M}$ be a TM. We can effectively construct a PCD $\mathcal{H}$ and an encoding $e : \Sigma^* \to \mathcal{Q}^d$ such that for any word w the following equivalence holds. $w \in L(\mathcal{M})$ iff $R^{\mathcal{H}}(e(w), O)$, where O denotes the origin.*

Corollary 2 (Computational power of (strongly non-zeno) PCD)

- *For any PCD $\mathcal{H}$ its reachability relation is r.e.*
- *Any r.e. set S is 1-reducible (see [15]) to the reachability relation of a PCD.*

2.5 Perturbed PCDs (PPCDs)

Consider a PCD $\mathcal{H}$ described by an ODE $\dot{\mathbf{x}} = f(\mathbf{x})$. For any $\varepsilon > 0$ the ε-perturbed system $\mathcal{H}_\varepsilon$ is described by the differential inclusion $||\dot{\mathbf{x}} - f(\mathbf{x})|| < \varepsilon$. This non-deterministic system can be considered as $\mathcal{H}$ submitted to a small noise with magnitude ε (see figure 3 (b)). We denote reachability in the system $\mathcal{H}_\varepsilon$ by $R_\varepsilon^{\mathcal{H}}(\cdot,\cdot)$. The limit reachability relation $R_\omega^{\mathcal{H}}(\mathbf{x},\mathbf{y})$ is introduced and an analog of Lemma 3 is stated exactly as for PAMs.

3 Results on PTMs

Our first result is that the ω-perturbed language of a TM is the complement of a recursively enumerable language.

Theorem 3 (Perturbed reachability is co-r.e.) *$L_\omega(\mathcal{M})$ is in the class Π_1^0.*

Proof: First, we show that for every $n \in \mathbb{N}$, $L_n(\mathcal{M})$ is a regular language:

Let us associate with the n-perturbed version of $\mathcal{M}$ a finite-state machine $A_\mathcal{M}$ defined as follows: (1) Each of its configurations is composed of a control state of $\mathcal{M}$ and a finite sequence of length $2n+1$ corresponding to the part of the configuration in the radius n from the head. There are $|Q| \times |\Sigma + 1|^{2n+1}$ such configurations. (2) The transition relation $\rightarrow$ is constructed by simulating the transitions of $\mathcal{M}$ and considering that, when the head is moved to the left (resp. to the right), a symbol in $\Sigma \cup \{B\}$ is nondeterministically chosen and appended to the left (resp. right) of the configuration and the right-most (resp. left-most) one is lost (it belongs now to the perturbed area of the configuration and hence it can be replaced by any other symbol).

To formulate the link between the computations of $A_\mathcal{M}$ and those of the n-perturbed version of $\mathcal{M}$ we need some definitions and notations: Let $Accept = (\Sigma \cup B)^n \times [F \times (\Sigma \cup B)] \times (\Sigma \cup B)^n$. Given a configuration of $\mathcal{M}$

$$c = \cdots a_{-n-1}a_{-n}a_{-n+1}\cdots a_{-1}[q,a_0]a_1\cdots a_{n-1}a_n a_{n+1}\cdots$$

we define the sequence

$$c|_n = a_{-n}a_{-n+1}\cdots a_{-1}[q,a_0]a_1\cdots a_{n-1}a_n$$

of length $2n+1$.

Then, it is easy to see that:

> *The n-perturbed version of $\mathcal{M}$ has an accepting run starting from a configuration c, if there exists $f \in Accept$ such that $c|_n \xrightarrow{*} f$ in $A_\mathcal{M}$.*

Hence, we can effectively construct $L_n(\mathcal{M})$ as a finite union of computable regular languages: Let Basis be the finite set of sequences $a_0a_1\cdots a_n \in \Sigma^{n+1}$ such that $B^n[q_{init},a_0]a_1\cdots a_n \xrightarrow{*} f$ for some $f \in Accept$. Let Short be the finite set of sequences $a_0a_1\cdots a_k \in \Sigma^*$ with $k < n$ such that $B^n[q_{init},a_0]a_1\cdots a_k B^{n-k} \xrightarrow{*} f$ for some $f \in Accept$. Then, we have

$$L_n(\mathcal{M}) = \text{Short} \cup \text{Basis}\Sigma^*$$

Since $L_n(\mathcal{M})$ is regular and effectively constructible, the same holds for its complement $\overline{L_n(\mathcal{M})}$. Hence, the set $\bigcup_n \overline{L_n(\mathcal{M})} = \overline{L_\omega(\mathcal{M})}$ is recursively enumerable as a union of a computable sequence of regular languages. □

A consequence of the theorem above is that *robust languages* (i.e. $L_\omega(\mathcal{M}) = L(\mathcal{M})$) are necessarily recursive (since they must be in $\Sigma_1^0 \cap \Pi_1^0$):

Corollary 3 (Robust $\Rightarrow$ decidable) *If $L_\omega(\mathcal{M}) = L(\mathcal{M})$ then $L(\mathcal{M})$ is recursive.*

The converse holds if we add another requirement on $\mathcal{M}$:

Proposition 1 (Decidable $\Rightarrow$ robust) *If $\mathcal{M}$ always stops (and hence $L(\mathcal{M})$ is recursive) then $L_\omega(\mathcal{M}) = L(\mathcal{M})$*

Now, we show that in general, ω-perturbed languages are not recursively enumerable. In fact, the following result says that some of them are complete among Π_1^0 languages.

Theorem 4 (Perturbed reachability is complete in Π_1^0) *For every TM $\mathcal{M}$, we can effectively construct another TM $\mathcal{M}'$ such that $L_\omega(\mathcal{M}') = \overline{L(\mathcal{M})}$.*

Proof: Let $\mathcal{M} = (Q, q_{init}, F, \Gamma)$ be a TM over Σ. Suppose w.l.o.g. that the machine $\mathcal{M}$ is such that, for every input $w \notin L(\mathcal{M})$, $\mathcal{M}$ never stops and uses an unbounded working space (the head goes arbitrarily far from the initial position).

Now, let us consider an extra symbol $\# \notin \Sigma$. Then, we define the TM $\mathcal{M}' = (Q', q'_{init}, F', \Gamma')$ over $\Sigma \cup \{\#\}$ as follows: $Q' = Q \cup \{q_f\}$, $q'_{init} = q_{init}$, $F' = \{q_f\}$, and $\Gamma' = \Gamma \cup \{(q,\#) \rightarrow (q_f,\#) : q \in Q\}$.

This means that $\mathcal{M}'$ is constructed as $\mathcal{M}$ except that all accepting states of $\mathcal{M}$ are rejecting for $\mathcal{M}'$ and that whenever $\mathcal{M}'$ sees the symbol $\#$, it stops in its unique accepting state q_f. Let us prove that we have indeed $L_\omega(\mathcal{M}') = \overline{L(\mathcal{M})}$.

Consider a word $w \in L(\mathcal{M})$. Then, there exists an accepting run of $\mathcal{M}$ on w. By definition of $\mathcal{M}'$, this run is rejecting for $\mathcal{M}'$. Let N be size of the space used by this run. It can be seen that the $(N+1)$-perturbed version of $\mathcal{M}'$ has exactly the same behavior as $\mathcal{M}'$ on w since perturbations in the non-visited part of the configuration

have no effect. Hence $w \notin L_{N+1}$, and consequently $w \notin L_\omega$ (Lemma 2).

Consider now a word $w \notin L(\mathcal{M})$. We show that for every $n > 0$, the n-perturbed version of $\mathcal{M}'$ recognizes w, which implies that w belongs to $L_\omega(\mathcal{M}')$. Let $n > 0$ and let us exhibit an accepting run of the n-perturbed version of $\mathcal{M}'$ on w: Suppose that, in the perturbed machine, starting from the initial configuration, two symbols at the distance $n+1$ to the left and to the right from the head are replaced by the symbol #. Then, since $w \notin L(\mathcal{M})$, the machine $\mathcal{M}$ has an unbounded run on w (see above the initial hypothesis on $\mathcal{M}$). Since $\mathcal{M}'$ has all the transitions of $\mathcal{M}$, it has also the same unbounded run on w, visiting positions arbitrarily far from the initial position of the head. Hence, the considered run of the n-perturbed version of $\mathcal{M}'$ eventually finds the # symbol and goes to the accepting state. □

4 Results on PPAMs

We consider now the case of perturbed PAMs and show that their perturbed reachability relation is co-recursively enumerable.

Theorem 5 (Perturbed reachability is co-r.e.) *The relation $R^{\mathcal{P}}_\omega(\mathbf{x},\mathbf{y})$ is Π^0_1 on $\mathcal{Q}^d$.*

Remember that in the case of TM, the proof of the similar result was based on the fact that the n-perturbed TM is in fact a finite-state system. For PAM, this actually does not hold, but we can show that each ε-perturbed PAM can be "faitfully " approximated by a finite-state automaton we define hereafter:

Consider a PAM $x := f(\mathbf{x}) = A_i\mathbf{x} + \mathbf{b}_i$ for $\mathbf{x} \in P_i$, $i = 1..N$. For any δ we can partition X into finitely many cubes $V_1, \ldots V_S$ of size δ. We say that V_j is a δ-successor of V_k if $dist(f(V_k), V_j) < \delta$, that is if some point of V_k can be mapped to a point near V_j. Now we can construct a finite automaton A_δ with states $Q_\delta = \{q_1, \ldots, q_S\}$, and with a transition from q_k to q_j authorized iff V_j is a δ-successor of V_k. Informally speaking, the automaton A_δ represents the PAM with accuracy δ. In order to formalize it we introduce the following *abstraction function* from X to Q_δ: $\alpha_\delta(\mathbf{x}) = q_i$ for $\mathbf{x} \in V_i$

Lemma 4 (Simulation) *(1) for any $\varepsilon > 0$ if $||f(\mathbf{x}) - \mathbf{y}|| < \varepsilon$ (i.e. the ε-perturbed system can make a transition from $\mathbf{x}$ to $\mathbf{y}$) then the automaton A_ε can make a transition from $\alpha_\varepsilon(\mathbf{x})$ to $\alpha_\varepsilon(\mathbf{y})$; (2) for any $\delta > 0$ if the automaton A_δ can make a transition from $\alpha_\delta(\mathbf{x})$ to $\alpha_\delta(\mathbf{y})$, then $||f(\mathbf{x}) - \mathbf{y}|| < C\delta$ (i.e. the $C\delta$-perturbed system can make a transition from $\mathbf{x}$ to $\mathbf{y}$), where C is a rational constant independent of δ;*

Proof: (1) Suppose that $||f(\mathbf{x}) - \mathbf{y}|| < \varepsilon$. Let $\alpha_\varepsilon(\mathbf{x}) = q_k$ and $\alpha_\varepsilon(\mathbf{y}) = q_j$. Then $dist(f(V_k), V_j) \leq dist(f(\mathbf{x}), \mathbf{y}) < \varepsilon$. Hence by definition of the automaton A_ε the state q_j is reachable from q_k.

(2) Suppose that $\alpha_\delta(\mathbf{x}) = q_k$ and $\alpha_\delta(\mathbf{y}) = q_j$ and the state q_j is reachable from q_k. In this case $dist(f(V_k), V_j) < \delta$. Hence there exist $\mathbf{x}_0 \in V_k$ and $\mathbf{y}_0 \in V_j$ such that $||f(\mathbf{x}_0) - \mathbf{y}_0|| < \delta$. As $\mathbf{x}_0$ and $\mathbf{x}$ are in the same cube V_k the distance between them is inferior to the diameter of this cube $\sqrt{d}\delta$. The same is true for $\mathbf{y}_0$ and $\mathbf{y}$. Finally

$$||f(\mathbf{x}) - \mathbf{y}|| \leq ||f(\mathbf{x}) - f(\mathbf{x}_0)|| + ||f(\mathbf{x}_0) - \mathbf{y}_0|| + \\ + ||\mathbf{y}_0 - \mathbf{y}|| < L\sqrt{d}\delta + \delta + \sqrt{d}\delta,$$

where the Lipschitz constant L can be found as $L = \max_i ||A_i||$. We can take now $C \geq L\sqrt{d} + 1 + \sqrt{d}$. □

Corollary 4 *$R^{\mathcal{P}}_\omega(\mathbf{x},\mathbf{y})$ holds iff for all rational $\delta > 0$ in the automaton A_δ the state $\alpha_\delta(\mathbf{y})$ is reachable from $\alpha_\delta(\mathbf{x})$.*

Hence by complementation $\neg R^{\mathcal{P}}_\omega(\mathbf{x},\mathbf{y})$ iff for some rational $\delta > 0$ the state $\alpha_\delta(\mathbf{y})$ is unreachable from $\alpha_\delta(\mathbf{y})$ in the automaton A_δ. Unreachability in this automaton is (uniformly in δ) decidable for any particular δ, and hence the relation $\neg R^{\mathcal{P}}_\omega$ is recursively enumerable, which terminates the proof of Theorem 5.

Corollary 5 (Robust ⇒ decidable) *If $R^{\mathcal{P}}_\omega = R^{\mathcal{P}}$ then $R^{\mathcal{P}}$ is recursive.*

Let us consider now the converse of Theorem 5. We prove the following fact:

Theorem 6 (Perturbed reachability is complete in Π^0_1) *Let $\mathcal{M}$ be a TM. We can effectively construct a PAM $\mathcal{P}$ and an encoding $e : \Sigma^* \to \mathcal{Q}^n$ such that for any word w, the following fact holds: $w \notin L(\mathcal{M})$ iff $R^{\mathcal{P}}_\omega(e(w), O)$.*

Proof: W.l.o.g. suppose that on any input word the machine $\mathcal{M}$ either stops in an accepting state, or computes forever. First we construct a 2-dimensional PAM $\mathcal{P}_0$ (and an input encoding $e : \Sigma^* \to \mathcal{Q}^n$) that simulates $\mathcal{M}$ and semi-recognizes $L(\mathcal{M})$ as described in [2]. Its main property is that for any word w the following equivalence holds: $w \in L(\mathcal{M})$ iff $R^{\mathcal{P}_0}_\omega(e(w), O)$. It is easy to verify that if a rather small neighborhood [1] (e.g. a 1/10-square) of the origin is reachable from $e(w)$ then $w \in L(\mathcal{M})$. The last useful property of this simulation is that all the points of the trajectory starting from $e(w)$ are internal points of polyhedra P_i.

[1] representing the accepting state of the TM

Now we construct a new 3-dimensional PAM $\mathcal{P}$ whose perturbed version will "semi-recognize" $\overline{L(\mathcal{M})}$. We will use notation $\mathbf{x}$ or $\mathbf{y}$ for 2-dimensional vectors and h for the third dimension (so the generic element of $\mathbb{R}^3$ will be $(\mathbf{x}, h)$). It is mainly the original system $\mathcal{P}_0$ embedded in the plane $h = 2$ of the space $\mathbb{R}^3$. However there are 2 changes (compare with the proof for TMs) — informally:

- The accepting state O (with his small neighborhood) of the original system $\mathcal{P}_0$ becomes rejecting for the new system $\mathcal{P}$.
- The zone $h \leq 1$ becomes accepting for the new system.

The idea is that for any $w \in L(\mathcal{M})$ the original PAM $\mathcal{P}_0$ will eventually arrive to O (and accept) and hence the perturbed PAM $\mathcal{P}$ will arrive to the neighborhood of $O \times \{2\}$ and reject. For any $w \notin L(\mathcal{M})$ the perturbed PAM $\mathcal{P}$ will slowly drift "down" until it reaches the accepting zone $h \leq 1$.

Formally, let the original system be defined on a subset of the cube $[-T, T]^2 \subseteq \mathbb{R}^2$ by equation $\mathbf{x} := f(\mathbf{x})$. Denote the squared neighborhood of the origin $[-0.1, 0.1]^2 \subseteq \mathbb{R}^2$ by C. Then the new system will be defined on the rectangular set $[-T-1, T+1]^2 \times [-1, 3] \subseteq \mathbb{R}^3$ by the equation $\mathbf{x} := g(\mathbf{x}, h)$ where $g(\mathbf{x}, h)$ is defined as follows:

- if $1 < h \leq 3$, and $\mathbf{x} \notin C$, then $g(\mathbf{x}, h) = (f(\mathbf{x}), h)$. Informally speaking, in the layer $1 < h < 3$ the system $\mathcal{P}$ simulates the original system $\mathcal{P}_0$ without modifying h
- if $1 < h \leq 3$ and $\mathbf{x} \in C$, then $g(\mathbf{x}, h)$ is undefined
- if $h \leq 1$ we go to the origin : $g(\mathbf{x}, h) = (\mathbf{0}, 0)$

The input encoding function for the system $\mathcal{P}$ is as follows: $e(w) = (e_0(w), 2)$ where e_0 is the encoding function of the original system $\mathcal{P}_t$.

Now we have to prove that $R_\omega^{\mathcal{P}}(e(w), O)$ iff $w \notin L(\mathcal{M})$. Suppose first that $w \notin L(\mathcal{M})$. In this case the TM $\mathcal{M}$ has an infinite-length run on w and the PAM $\mathcal{P}_0$ has an infinite trajectory $\mathbf{x}_n$ starting in $e_0(w)$. For any $\varepsilon > 0$ we can construct a trajectory g of the ε-perturbed system $\mathcal{P}$ as follows:

- $g_n = (\mathbf{x}_n, 2 - \varepsilon n)$ for $n \in [0, \lceil 1/\varepsilon \rceil]$; during the first $\lceil 1/\varepsilon \rceil$ time units the system simulates $\mathcal{P}_0$ along first two dimensions slowly drifting down in the third one
- $g_n = 0$ for $n \geq \lceil 1/\varepsilon \rceil$ the trajectory jumps to the origin and stays there.

It is easy to see that g_n is a trajectory of the ε-perturbed system, and hence $R_\omega^{\mathcal{P}}(e(w), O)$ holds.

Now consider the other case when $w \notin L(\mathcal{M})$. Then the trajectory $\mathbf{x}_n$ of $\mathcal{P}_0$ starting in $e_0(w)$ eventually arrives to the origin. The non-perturbed trajectory g_n of $\mathcal{P}$ starting in $e(w)$ will follow $\mathbf{x}_n$ in the plane $h = 2$ until it reaches the neighborhood C of the origin. Once in this neighborhood the system $\mathcal{P}$ dies immediately. The only thing to verify is that all perturbed trajectories of $\mathcal{P}$ starting in $e(w)$ are close enough to g_n for ε small enough. Let T be the time of arrival to the origin (i.e. such that $g_T = 0$), $A = \max\{1, ||A_i||\}$ and $\theta = \min_{n<T} dist(\mathbf{x}_n, \partial P_{i(n)})$. If we take $\varepsilon < \theta A^{-T}$, then a straightforward induction shows that any ε-perturbed trajectory g'_n is close to g_n and the same affine maps are applied until it enters the deadly neighborhood of the origin. □

Theorem 7 *All the results stated in this section can be proved in a very similar manner for Linear hybrid automata (LHA).*

5 Results on PPCDs

We consider finally the case of PCDs and prove the same results as for PAMs (and LHAs). The overall structure of the proofs is the same as in the previous case. However, the proofs for the two kinds of models are technically different due to the fact that the rules for accumulating errors (resulting from perturbations) are different for each of these models. An ε-perturbation of a PAM results in moving the state by ε in any direction at each transition, which ensures the simulation lemma 4 (the same holds in the LHA model). Differently from this, a perturbed trajectory in an ε-perturbed PCD deviates from the ideal trajectory after crossing a region by $\sim \tau\varepsilon$, where τ stands for the time needed to cross this region, and this time depends on the entry point to a region and the slope at this region and cannot be bounded from below.

Our solution to this consists in observing (and approximating by an automaton) the states of the PCD only when it enters some special *good regions*. In a non-Zeno system, the time τ' between consecutive visits of good regions is bounded from below and the accumulated error $\sim \tau'\varepsilon$ is large enough to ensure simulation.

Theorem 8 (Perturbed reachability is co-r.e.) *The relation $R_\omega^{\mathcal{H}}(\mathbf{x}, \mathbf{y})$ on $\mathbb{Q}^d$ is in Π_1^0.*

We proceed in a similar manner as for PAMs: We approximate the ε-perturbed system by a finite-state automaton. However, relations between the system and the automaton are somewhat subtler. First of all, let N be the number of regions in the PCD, and $\alpha > 0$ a positive constant specified below. Without loss of generality we can suppose that the

norm used in the definition of ε-perturbed system is $\|\cdot\|_\infty$, which means that ε-ball centered in a point$\mathbf{x}$ is in fact a cube with side 2ε. Let us introduce now some definitions:

Definition 3 (Good points) *A point* $\mathbf{x}$ *on the boundary of a region is* good *if the trajectory starting from* $\mathbf{x}$ *does not change direction during at least* α *time. Formally let* $\mathbf{c} = f(\mathbf{x})$ *be the slope in* $\mathbf{x}$*. Then the vector field* $f(\mathbf{y})$ *should be constant (and equal to* $\mathbf{c}$*) for all* $\mathbf{y} \in [\mathbf{x}, \mathbf{x} + \alpha\mathbf{c}]$

Lemma 5 (Good regions) *The set* G *of all good points is a finite union of polyhedra of dimensionality* $< d$.

The following lemma, saying that the good regions are visited often, enough follows from the strong non-zenoness of the PCD.

Lemma 6 *Each perturbed trajectory crossing* N *regions visits a good region at least once.*

Let us see now how we define an "approximating automaton": For any δ we can partition G into finitely many polyhedra $V_1, \ldots V_S$ of size δ. We say that V_j is a δ-successor of V_k if there exists a trajectory of the δ-perturbed system no more than N links from an $\mathbf{x} \in V_k$ to an $\mathbf{y} \in V_j$. It is easy to see that the property of being a δ-successor can be reduced to a linear programming problem, and hence is decidable.

Then, we can construct a finite automaton A_δ with states $Q_\delta = \{q_1, \ldots, q_S\}$, and with a transition from q_k to q_j authorized iff V_j is a δ-successor of V_k. Informally speaking, the automaton A_δ represents the δ-perturbed PCD with accuracy δ. In order to formalize it we introduce the following *abstraction function* from X to Q_δ: $\alpha_\delta(\mathbf{x}) = q_i$ for $\mathbf{x} \in V_i$.

Hereafter, we explore in which sense A_δ simulates $\mathcal{H}_\varepsilon$:

Lemma 7 (Quasi-Simulation) *Let* $\mathbf{x}, \mathbf{y} \in G$ *be two good points. (1) for any* $\varepsilon > 0$ *if the* ε*-perturbed system can go from* $\mathbf{x}$ *to* $\mathbf{y}$ *via a trajectory with less than* N *links, then the automaton* A_ε *can make a transition from* $\alpha_\varepsilon(\mathbf{x})$ *to* $\alpha_\varepsilon(\mathbf{y})$*; (2) for any* $\delta > 0$ *if the automaton* A_δ *can make a transition from* $\alpha_\delta(\mathbf{x})$ *to* $\alpha_\delta(\mathbf{y})$*, then* $C\delta$*-perturbed system can go from* $\mathbf{x}$ *to a good point* $\mathbf{y}'$ *via a trajectory with less than* N *links, where* C *is a rational constant independent of* δ*, and* $\alpha_\delta(\mathbf{y}) = \alpha_\delta(\mathbf{y}')$*;*

Corollary 6 (Many steps) *Let* $\mathbf{x}, \mathbf{y} \in G$ *be two good points. (1) for any* $\varepsilon > 0$ *if the* ε*-perturbed system has a trajectory from* $\mathbf{x}$ *to* $\mathbf{y}$ *, then the automaton* A_ε *has a run from* $\alpha_\varepsilon(\mathbf{x})$ *to* $\alpha_\varepsilon(\mathbf{y})$*; (2) for any* $\delta > 0$ *if the automaton* A_δ *has a run from* $\alpha_\delta(\mathbf{x})$ *to* $\alpha_\delta(\mathbf{y})$*, then* $C\delta$*-perturbed system has a trajectory from* $\mathbf{x}$ *to a good point* $\mathbf{y}'$*), where* $\alpha_\delta(\mathbf{y}) = \alpha_\delta(\mathbf{y}')$.

It is still not the result that we want, because first it concerns only reachability between good points, and, second, the target point $\mathbf{y}$ is replaced by a neighbor point $\mathbf{y}'$.

In order to deal with these two issues we introduce the following δ-test for perturbed reachability between arbitrary points. First of all we construct the A_δ automaton. Next, we proceed in three steps:

1. Find the set S_1 of indices i such that V_i is reachable by $\mathcal{H}_\delta$ from $\mathbf{x}$ via a trajectory with less then N links. This can be done algorithmically using linear programming.

2. Find the set S_2 of indices of all the states q_j of the A_δ automaton reachable in this automaton from $\{q_i \mid i \in S_1\}$. This is a reachability problem in a finite-state automaton.

3. For each $j \in S_2$ test whether $\mathbf{y}$ is reachable by $\mathcal{H}_\delta$ from V_j via a trajectory with less then N links. This can be solved as in the first step using linear programming. In case of positive answer for any $j \in S_2$, the δ-test succeeds, otherwise it fails.

Notice that δ-test always terminates. Then, it is easy to see that the following fact holds:

Lemma 8 (Correctness of δ-test) *For any two points* $\mathbf{x}$ *and* $\mathbf{y}$ *(1) if* $R^{\mathcal{H}}_\varepsilon(\mathbf{x}, \mathbf{y})$*, then* δ*-test succeeds for* $\mathbf{x}$ *and* $\mathbf{y}$*. (2) If* δ*-test succeeds for* $\mathbf{x}$ *and* $\mathbf{y}$*, then* $R^{\mathcal{H}}_{C\delta}(\mathbf{x}, \mathbf{y})$.

Corollary 7 $(\mathbf{x}, \mathbf{y}) \notin R^{\mathcal{H}}_\omega$ *if and only if for some* $n \in \mathbb{N}$ *the* $1/n$*-test fails for* $\mathbf{x}$ *and* $\mathbf{y}$.

By the corollary above, a semi-decision algorithm for $\neg R^{\mathcal{H}}_\omega$ is immediate, which terminates the sketch of proof of Theorem 8.

Corollary 8 (Robust $\Rightarrow$ decidable) *If* $R^{\mathcal{H}}_\omega = R^{\mathcal{H}}$ *then* $R^{\mathcal{H}}$ *is recursive.*

Finally, we can prove the converse result of Theorem 8. The proof is given in the appendix.

Theorem 9 (Perturbed reachability is complete in Π^0_1) *Let* $\mathcal{M}$ *be a TM. We can effectively construct a PCD* $\mathcal{H}$ *and an encoding* $e : \Sigma^* \to \mathcal{Q}^n$ *such that for any word* w *the following equivalence holds:* $w \notin L(\mathcal{M})$ *iff* $R^{\mathcal{H}}_\omega(e(w), O)$.

6 Conclusion

We have shown that when we consider infinitesimal perturbations in the dynamics of a system, the reachability relation becomes co-recursively enumerable, which proves that robust systems are decidable. It is interesting to observe that these results hold for several different discrete

and continuous time models of dynamic systems, which shows that they correspond to a general phenomenon. The proofs of these results have also a common scheme, although they differ significantly depending from the specificity of the dynamics of each class of models.

Our results establish a tight link between the notions of decidability and robustness for infinitesimal perturbations. This link is of a semantical nature. An interesting question is to find sufficient "syntactical" conditions on the models of dynamical systems ensuring their robustness, leading to decidability results for classes of dynamical systems.

References

[1] Cris Moore, "Generalized shifts: Undecidability and unpredictability in dynamical systems," *Nonlinearity*, vol. 4, pp. 199–230, 1991.

[2] Pascal Koiran, Michel Cosnard, and Max Garzon, "Computability with low-dimensional dynamical systems," *Theoretical Computer Science*, vol. 132, pp. 113–128, 1994.

[3] Eugene Asarin, Oded Maler, and Amir Pnueli, "Reachability analysis of dynamical systems having piecewise-constant derivatives," *Theoretical Computer Science*, vol. 138, pp. 35–65, 1995.

[4] Bruce Krogh and Nancy Lynch, Eds., *Hybrid Systems: Computation and Control (HSCC 2000)*, vol. 1790 of *LNCS*. Springer, 2000.

[5] Thomas A. Henzinger, P.W. Kopke, Anuj Puri, and Pravin Varaiya, "What's decidable about hybrid automata?," in *Proceedings of the 27th Annual Symposium on Theory of Computing*. 1995, pp. 373–382, ACM Press.

[6] Martin Fränzle, "Analysis of hybrid systems: An ounce of realism can save an infinity of states," in *Computer Science Logic (CSL'99)*, Jörg Flum and Mario Rodríguez-Artalejo, Eds. 1999, vol. 1683 of *LNCS*, pp. 126–140, Springer.

[7] Thomas A. Henzinger and Jean-François Raskin, "Robust undecidability of timed and hybrid systems," In Krogh and Lynch [4], pp. 145–159.

[8] E. Asarin, "Chaos and Undecidability," Tech. Rep., Verimag, 1995.

[9] Peter Kloeden and Victor Kozyakin, "The inflation of attractors and discretization: the autonomous case," *Nonlinear Anal., TMA*, vol. 40, pp. 333–343, 2000.

[10] Anuj Puri, "Dynamical properties of timed automata," *Discrete Event Dynamic Systems*, vol. 10, no. 1/2, pp. 87–113, 2000.

[11] M. Casey, "The dynamics of discrete-time computation, with application to recurrent neural networks and finite-state machine extraction," *Neural Computation*, vol. 8:6, 1996.

[12] W. Maass and P. Orponen, "On the effect of analog noise in discrete-time analog computations," in *Neural Information Processing Systems*, 1996.

[13] C. Moore, "Finite-dimensional analog computers: Flows, maps, and recurrent neural networks," in *1st Intern. Conf. on Unconventional Models of Computation*. 1998, Springer-Verlag.

[14] C. Moore, "Undecidability and unpredictability in dynamical systems," *Physical Review Letters*, vol. 64, pp. 2354–2357, 1990.

[15] Hartley Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

A Proof of Theorem 9

The idea of this proof is similar to the case of PAMs (Theorem 6). We take a PCD $\mathcal{H}_0$ simulating the machine $\mathcal{M}$, and add one more dimension h. We start at the level $h = 4$. Accepting states of the PCD $\mathcal{H}_0$ become rejecting in the new PCD $\mathcal{H}$. In order to be accepting in $\mathcal{H}$ the trajectory should go down and reach the plane $h = 0$. It is possible for arbitrarily small ϵ only if the original PCD $\mathcal{H}_0$ can evolve during arbitrarily long time, that is the perturbed version of $\mathcal{H}$ accepts a word iff $\mathcal{H}_0$ does not accept it.

First let us construct a 4-dimensional PCD $\mathcal{H}_0$ (and an input encoding $e : \Sigma^* \to \mathbb{Q}^n$) which simulates $\mathcal{M}$ and semi-recognizes $L(\mathcal{M})$ as described in [3]. Its main property is that for any word w the following equivalence holds. $w \in L(\mathcal{M})$ if and only if $R_\omega^{\mathcal{H}_0}(e(w), O)$ It is easy to verify that if a rather small neighborhood (e.g. a 1/10-ball) of the origin is reachable from $e(w)$ then $w \in L(\mathcal{M})$.

Now we construct a new 5-dimensional PCD $\mathcal{H}$ whose perturbed version will "semi-recognize" $\overline{L(\mathcal{M})}$. We will use notation $\mathbf{x}, \mathbf{y}$ for 4-dimensional vectors and h for the fifth dimension (so the generic element of $\mathbb{R}^5$ will be $(\mathbf{x}, h)$). It is mainly the original system $\mathcal{H}_0$ submerged in the hyperplane $h = 3$ of the space $\mathbb{R}^5$. However there are 2 changes (compare with the proof for PAMs) — informally:

- The accepting state O (with his small neighborhood) of the original system $\mathcal{H}_0$ becomes rejecting for the new system $\mathcal{H}$
- The zone $h \leq 1$ becomes accepting for the new system

The idea is that for any $w \in L(\mathcal{M})$ the original PCD $\mathcal{H}_0$ will eventually arrive to O (and accept) and hence the

perturbed PCD $\mathcal{H}$ will arrive to the neighborhood of $O \times 2$ and reject. For any $w \notin L(\mathcal{M})$ the perturbed PCD $\mathcal{H}$ will slowly drift "down" until it reaches the accepting zone $h \leq 1$.

Formally, let the original system be defined on a subset of the cube $[-T,T]^4 \subseteq I\!R^4$ by equation $\dot{\mathbf{x}} = f(\mathbf{x})$. Denote the cubic neighborhood of the origin $[-0.1,0.1]^4 \subseteq I\!R^4$ by C.

Then the new system will be defined on the rectangular set $[-T-1,T+1]^4 \times [-1,5] \subseteq I\!R^5$ by the equation $(\mathbf{x},h)' = g(\mathbf{x},h)$ where $g(\mathbf{x},h)$ is defined as follows:

- if $h \geq 4$, then $g(\mathbf{x},h) = (\mathbf{0},1)$: anything that arrives in the layer $h \geq 4$ goes "up" and is rejected
- if $2 < h < 4$ and $f(\mathbf{x})$ is defined, then $g(\mathbf{x},h) = (f(\mathbf{x}),0)$. Informally speaking, in the layer $2 < h < 4$ the system $\mathcal{H}$ simulates the original system $\mathcal{H}_0$
- if $2 < h < 4$ and $\mathbf{x} \in C$, then $g(\mathbf{x},h) = (\mathbf{0},1)$
- if $2 < h < 4$ and $f(\mathbf{x})$ is undefined, then $g(\mathbf{x},h) = (\mathbf{0},1)$
- if $1 < h \leq 2$ we go down : $g(\mathbf{x},h) = (\mathbf{0},-1)$
- finally in the layer $-1 \leq h < 1$ we put a (piecewise constant) vector field with all the trajectories going to the origin.

The input encoding function for the system $\mathcal{H}$ is as follows: $e(w) = (e_0(w),3)$ where e_0 is the encoding function of the original system $\mathcal{H}_0$.

Now we have to prove that $R_\omega^{\mathcal{H}}(e(w),O)$ if and only if not $w \notin L(\mathcal{M})$. Suppose first that $w \notin L(\mathcal{M})$. In this case the TM $\mathcal{M}$ has an infinite-length run on w and the PCD $\mathcal{H}_0$ has an infinite trajectory $\mathbf{x}(t)$ starting in $e_0(w)$. For any $\varepsilon > 0$ we can construct a trajectory g of the ε-perturbed system $\mathcal{H}$ as follows:

- $g(t) = (\mathbf{x}(t), 3 - \varepsilon t)$ for $t \in [0, 1/\varepsilon]$; during the first $1/\varepsilon$ time units the system simulates $\mathcal{H}_0$ along first four dimensions slowly drifting down in the fifth one
- $g(t) = (\mathbf{x}(1/\varepsilon), 2-(t-1/\varepsilon))$ for $t \in [1/\varepsilon; 1/\varepsilon+1]$— the next trajectory segment goes straight down with unit velocity during one time unit.
- The last trajectory segment goes straight to the origin.

Now consider the other case when $w \notin L(\mathcal{M})$. Then the trajectory $\mathbf{x}(t)$ of $\mathcal{H}_0$ starting in $e_0(w)$ eventually arrives to the origin. The non-perturbed trajectory $g(t)$ of $\mathcal{H}$ starting in $e(w)$ will follow $\mathbf{x}(t)$ in the plane $h = 3$ until it reaches the neighborhood C of the origin. Once in this neighborhood the system $\mathcal{H}$ goes straight up to the death. The only thing to verify is that all perturbed trajectories of $\mathcal{H}$ starting in $e(w)$ are close enough to $g(t)$ for ε small enough. This can be done similarly to PAMs.

From Verification to Control: Dynamic Programs for Omega-regular Objectives*

Luca de Alfaro Thomas A. Henzinger Rupak Majumdar

Electrical Engineering and Computer Sciences, University of California, Berkeley
{dealfaro,tah,rupak}@eecs.berkeley.edu

Abstract. Dynamic programs, or fixpoint iteration schemes, are useful for solving many problems on state spaces, including model checking on Kripke structures ("verification"), computing shortest paths on weighted graphs ("optimization"), computing the value of games played on game graphs ("control"). For Kripke structures, a rich fixpoint theory is available in the form of the μ-calculus. Yet few connections have been made between different interpretations of fixpoint algorithms. We study the question of when a particular fixpoint iteration scheme φ for verifying an ω-regular property Ψ on a Kripke structure can be used also for solving a two-player game on a game graph with winning objective Ψ. We provide a sufficient and necessary criterion for the answer to be affirmative in the form of an *extremal-model theorem for games*: under a game interpretation, the dynamic program φ solves the game with objective Ψ if and only if both (1) under an existential interpretation on Kripke structures, φ is equivalent to $\exists\Psi$, and (2) under a universal interpretation on Kripke structures, φ is equivalent to $\forall\Psi$. In other words, φ is correct on all two-player game graphs iff it is correct on all extremal game graphs, where one or the other player has no choice of moves. The theorem generalizes to quantitative interpretations, where it connects two-player games with costs to weighted graphs.

While the standard translations from ω-regular properties to the μ-calculus violate (1) or (2), we give a translation that satisfies both conditions. Our construction, therefore, yields fixpoint iteration schemes that can be uniformly applied on Kripke structures, weighted graphs, game graphs, and game graphs with costs, in order to meet or optimize a given ω-regular objective.

*This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the AFOSR MURI grant F49620-00-1-0327, the NSF Theory grant CCR-9988172, and the NSF ITR grant CCR-0085949.

1 Introduction

If Ψ is a property of a Kripke structure, then every μ-calculus formula φ that is equivalent to Ψ prescribes an algorithm for model checking Ψ. This is because the μ-calculus formula φ can be computed by iterative fixpoint approximation. Indeed, the μ-calculus has been called the "assembly language" for model checking.

In control, we are given a two-player game structure and an objective, and we wish to find out if player 1 (the "controller") has a strategy such that for all strategies of player 2 (the "plant") the outcome of the game meets the objective. If the outcome of a game is an infinite sequence of states, then objectives are naturally specified as ω-regular properties [15]. A simple but important objective is the reachability property $\Diamond T$, for a set T of states, which asserts that player 1 wins if it can direct the game into the target set T, while player 2 wins if it can prevent the game from entering T forever. We write $\langle\langle 1\rangle\rangle\Diamond T$ for the reachability game with target T for player 1. A dynamic program for solving the reachability game can be viewed as evaluating a fixpoint equation, namely,

$$\langle\langle 1\rangle\rangle\Diamond T \;=\; \mu x.(T \vee \mathit{1Pre}(x)),$$

where $\mathit{1Pre}(T)$ is the set of states from which player 1 can force the game into T in a single step. It is not difficult to see that this fixpoint equation is identical to the μ-calculus expression for model checking the reachability property $\exists\Diamond T$, namely,

$$\exists\Diamond T \;=\; \mu x.(T \vee \mathit{EPre}(x)), \qquad (1)$$

except for the use of the *predecessor operator* EPre in place of $\mathit{1Pre}$, where $\mathit{EPre}(T)$ is the set of states that have a successor in T.

For every ω-regular property Ψ, it is well-known how to construct an equivalent μ-calculus formula φ

0-7695-1281-X/01 $10.00 © 2001 IEEE

[7, 4], which can then be used to model check $\exists\Psi$, i.e., to compute the set of states from which there is a path satisfying Ψ. Now suppose we want to solve the control problem with objective Ψ. The question we set out to answer in this paper is whether φ is of any use for this purpose; more specifically, if we simply replace all *EPre* operators in φ by *1Pre* operators, do we obtain an algorithm for solving the game with objective Ψ, i.e., for computing the set of states from which player 1 can ensure that Ψ holds?

In general, the answer is negative. Consider the co-Büchi property $\Diamond\Box T$, which asserts that, eventually, the target T is reached and never left again. The Emerson-Lei translation [7] yields the equivalent μ-calculus formula

$$\exists\Diamond\Box T \;=\; \mu x.(\mathit{EPre}(x) \vee (\nu y.\mathit{EPre}(y) \wedge T)). \quad (2)$$

The Dam translation [4] gives

$$\exists\Diamond\Box T \;=\; \mu x.(\mathit{EPre}(x) \vee (T \wedge \mathit{EPre}(\nu y.(T \wedge \mathit{EPre}(y))))), \quad (3)$$

and Bhat-Cleaveland [2] produce the same result. But neither of these formulas give the correct solution for games. To see this, consider the following game on the state space $\{s_1, s_2, s_3\}$. At s_1, player 2 can play two moves: one of them keeps the game in s_1, the other takes the game to s_2. At s_2, player 1 can play two moves: one of them keeps the game in s_2, the other takes the game to s_3. Once in s_3, the game remains in s_3 forever. The target set is $T = \{s_1, s_3\}$. Then, $\langle\langle 1\rangle\rangle\Diamond\Box T = \{s_1, s_2, s_3\}$. However, both equations (2) and (3) denote the smaller set $\{s_2, s_3\}$ when *EPre* is replaced by *1Pre*.

We present an extremal-model theorem which says that the fixpoint formula φ over *1Pre* solves the game with ω-regular objective Ψ if and only if both of the following conditions are met:

- **E** The *EPre* version of φ is equivalent to the existential property $\exists\Psi$.
- **A** The *APre* version of φ is equivalent to the universal property $\forall\Psi$. (Here, $\mathit{APre}(T)$ is the set of states all of whose successors lie in T, and $\forall\Psi$ holds at a state if all paths from the state satisfy Ψ.)

In other words, for a fixpoint formula φ to solve the game with ω-regular objective Ψ, it is not only necessary but also sufficient that φ coincides with Ψ under the two extremal, non-game interpretations. In the co-Büchi example, while the expressions (2) and (3) satisfy condition **E** of the extremal-model theorem, they violate condition **A**. By contrast, in the reachability example, the expression (1) meets also condition **A**, because

$$\forall\Diamond T \;=\; \mu x.(T \vee \mathit{APre}(x)).$$

We show constructively that for every ω-regular objective Ψ there is indeed a fixpoint formula φ which meets both conditions of the extremal-model theorem. The construction is based on the determinization of ω-automata [12, 13], and on the translation from alternating ω-automata to μ-calculus [5]. In particular, for the co-Büchi property we obtain

$$\langle\langle 1\rangle\rangle\Diamond\Box T \;=\; \mu x.\nu y.(\mathit{1Pre}(x) \vee (\mathit{1Pre}(y) \wedge T)). \quad (4)$$

The reader may check that both

$$\exists\Diamond\Box T \;=\; \mu x.\nu y.(\mathit{EPre}(x) \vee (\mathit{EPre}(y) \wedge T)),$$
$$\forall\Diamond\Box T \;=\; \mu x.\nu y.(\mathit{APre}(x) \vee (\mathit{APre}(y) \wedge T)).$$

In general, our translation provides optimal algorithms for solving games with ω-regular objectives: in particular, if the objective is given by a formula Ψ of linear temporal logic, then the resulting algorithm has a 2EXPTIME complexity in the length of Ψ [11].

Our results also shed light on a related question: given a "verification" μ-calculus formula φ_v that uses only the predecessor operator *EPre*, what is the relation between φ_v and its "control" version φ_c, obtained by replacing *EPre* with *1Pre*? From [6] we know that if φ_v is *deterministic*, i.e., if every conjunction in φ_v has at least one constant argument, then φ_v specifies an ω-regular language; that is, φ_v is equivalent to $\exists\Psi$ for some ω-regular property Ψ. We introduce the syntactic class of *strongly deterministic* μ-calculus formulas, a subclass of the deterministic formulas, and we show that if φ_v is strongly deterministic, then φ_v solves the verification problem for specification $\exists\Psi$ iff φ_c solves the control problem for objective Ψ. This correspondence does not hold in general for deterministic formulas.

We extend the connection between verification and control also to *quantitative* properties. Consider a graph with nonnegative edge weights, which represent costs. By defining an appropriate quantitative predecessor operator Pre_+, the dynamic program for reachability, $\mu x.(T \vee \mathit{Pre}_+(x))$, computes the cost of the shortest path to the target T. Similarly, consider a game whose moves incur costs. Then again, for a suitable quantitative predecessor operator $\mathit{1Pre}_+$, the dynamic program $\mu x.(T \vee \mathit{1Pre}_+(x))$ computes the real value of the game, which is defined as the minimal cost for player 1 to reach the target T (or infinity, if player 1 has no strategy to reach T). For general ω-regular objectives, we define the cost of the infinite

outcome of a game as the cost of the shortest (possibly finite) prefix that is a witness to the objective. We show that the extremal-model theorem applies to this quantitative setting also. This gives us dynamic programs for solving the real-valued games with respect to all ω-regular objectives. For example, equation (4) with *1Pre* replaced by $1Pre_{\mathbb{R}}$ specifies a dynamic program for the quantitative co-Büchi game, whose value is the minimal cost for player 1 to reach and stay inside the target T (this cost is infinite unless player 1 can enforce an infinite sequence of moves all but finitely many of which have cost 0).

2 Reachability and Safety

We define our setting, and in doing so, review some well-known results about iterative solutions for simple verification, optimization, and control problems, where the objective is to reach or avoid a given set of states (expending minimal cost).

2.1 Game structures

We define game structures over a global set A of *actions*, and a global set P of *propositions*. A (two-player) *game structure* $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$ (over A and P) consists of a finite set S of *states*, two action assignments Γ_1, Γ_2: $S \to 2^A \setminus \emptyset$ which define for each state two nonempty, finite sets of actions available to player 1 and player 2, respectively, a *transition function* δ: $S \times A \times A \to S$ which associates with each state s and each pair of actions $a \in \Gamma_1(s)$ and $b \in \Gamma_2(s)$ a successor state, a *weight function* w: $S \times A \times A \to \mathbb{R}_{\geq 0}$ which associates with each state s and each pair of actions $a \in \Gamma_1(s)$ and $b \in \Gamma_2(s)$ a nonnegative real, and a proposition assignment $\langle\cdot\rangle$: $S \to 2^P$ which defines for each state s a finite set $\langle s\rangle \subseteq P$ of propositions that are true in s. Intuitively, at state s, player 1 chooses an action a from $\Gamma_1(s)$ and, simultaneously and independently, player 2 chooses an action b from $\Gamma_2(s)$. Then, the game proceeds to the successor state $\delta(s, a, b)$. The nonnegative real $w(s, a, b)$ represents the "cost" of the transition $\delta(s, a, b)$ (if it is to be minimized), or a "reward" (if it is to be maximized). Given a proposition $p \in P$, a state $s \in S$ is called a *p-state* iff $p \in \langle s\rangle$. If S is not given explicitly, then we write S^G to denote the state space of the game structure G.

Game structures are "concurrent" [1]; they subsume "turn-based" game structures (i.e., and-or graphs), where in each state at most one of the two players has a choice of actions. A special case of turn-based games are the one-player structures. A *one-player structure* is either a player-1 structure or a player-2 structure. The game structure G is a *player-1 structure* if $\Gamma_2(s)$ is a singleton for all states $s \in S$; and G is a *player-2 structure* if $\Gamma_1(s)$ is a singleton for all $s \in S$. In player-1 structures, player 2 has no choices, and in player-2 structures, player 1 has no choices. Every game structure G defines an *underlying transition structure* $K^G = (S, \to, \langle\cdot\rangle)$, where for all states $s, t \in S$, we have $s \to t$ iff there exist actions $a \in \Gamma_1(s)$ and $b \in \Gamma_2(S)$ such that $\delta(s, a, b) = t$. Transition structures do not distinguish between individual players.

Restrictions of game structures. A *player-1 restriction* of the game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$ is a game structure of the form $G_1 = (S, \Gamma'_1, \Gamma_2, \delta, \langle\cdot\rangle)$ with $\Gamma'_1(s) \subseteq \Gamma_1(s)$ for all states $s \in S$. Symmetrically, a *player-2 restriction* of G is a game structure of the form $G_2 = (S, \Gamma_1, \Gamma'_2, \delta, \langle\cdot\rangle)$ with $\Gamma'_2(s) \subseteq \Gamma_2(s)$ for all $s \in S$. In other words, for $i = 1, 2$, a player-i restriction of a game structure restricts the action choices that are available to player i.

Strategies and runs. Consider a game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$. A *player-i strategy*, for $i = 1, 2$, is a function ξ_i: $S^+ \to A$ that maps every nonempty, finite sequence of states to an action available to player i at the last state of the sequence; that is, $\xi_i(\overline{s} \cdot s) \in \Gamma_i(s)$ for every state sequence $\overline{s} \in S^*$ and every state $s \in S$. Intuitively, $\xi_i(\overline{s} \cdot s)$ indicates the choice taken by player i according to strategy ξ_i if the current state of the game is s, and the history of the game is $\overline{s}$. We write Ξ_i for the set of player-i strategies. We distinguish the following types of strategies. The strategy ξ_i is *memoryless* if in every state $s \in S$, the choice of player i depends only on s; that is, $\xi_i(\overline{s} \cdot s) = \xi_i(s)$ for all state sequences $\overline{s} \in S^*$. The strategy ξ_i is *finite-memory* if in every state $s \in S$, the choice of player i depends only on s, and on a finite number of bits about the history of the game; the formal definition is standard [5].

A *run* r of the game structure G is a nonempty, finite or infinite sequence $s_0(a_0, b_0)s_1(a_1, b_1)s_2 \ldots$ of alternating states $s_j \in S$ and action pairs $(a_j, b_j) \in \Gamma_1(s_j) \times \Gamma_2(s_j)$ such that $s_{j+1} = \delta(s_j, a_j, b_j)$ for all $j \geq 0$. The first state s_0 is called the *source* of the run. The *weight* of the run is $w(r) = \Sigma_{j \geq 0} w(s_j, a_j, b_j)$; the weight $w(r)$ is either a real number, or infinity (if the sum diverges). Let $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$ be a pair of strategies for player 1 and player 2, respectively. The *outcome* $R_{\xi_1, \xi_2}(s)$ from state $s \in S$ of the strategies ξ_1 and ξ_2 is a source-s infinite run of G, namely, $R_{\xi_1, \xi_2}(s) = s_0(a_0, b_0)s_1(a_1, b_1)s_2 \ldots$ such that (1) $s_0 = s$ and (2) for all $j \geq 0$, both $a_j = \xi_1(s_0 s_1 \cdots s_j)$ and $b_j = \xi_2(s_0 s_1 \cdots s_j)$.

$$\begin{Bmatrix} 1Pre_{\mathbb{B}}^G \\ 2Pre_{\mathbb{B}}^G \end{Bmatrix}(f)(s) = \begin{Bmatrix} \exists a \in \Gamma_1(s).\forall b \in \Gamma_2(s) \\ \exists b \in \Gamma_2(s).\forall a \in \Gamma_1(s) \end{Bmatrix}. f(\delta(s,a,b)) \quad (5)$$

$$\begin{Bmatrix} EPre_{\mathbb{B}}^G \\ APre_{\mathbb{B}}^G \end{Bmatrix}(f)(s) = \begin{Bmatrix} \exists a \in \Gamma_1(s).\exists b \in \Gamma_2(s) \\ \forall a \in \Gamma_1(s).\forall b \in \Gamma_2(s) \end{Bmatrix}. f(\delta(s,a,b)) \quad (6)$$

Figure 1: Boolean game and transition predecessor operators

2.2 Single-step verification and control

Values and valuations. A *value lattice* is a complete lattice $(V, \sqcup, \sqcap, \top, \bot)$ of *values* V with join $\sqcup$, meet $\sqcap$, top element $\top$, and bottom element $\bot$. Given $u, v \in V$, we write $u \sqsubseteq v$ iff $u = u \sqcap v$. Consider a game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$. A *valuation* f for G on the value lattice V is a function from states to values; that is, f: $S \to V$. The set $[S \to V]$ of valuations is again a lattice, with the lattice operations $(\sqcup, \sqcap, \top, \bot)$ defined pointwise; for example, for two valuations f_1 and f_2, we have $(f_1 \sqcup f_2)(s) = f_1(s) \sqcup f_2(s)$ for all states $s \in S$. If f: $S \to V$ is a valuation such that $f(s) \in \{\top, \bot\}$ for all states $s \in S$, then by $-f$ we denote the "complementary" valuation with $-f(s) = \top$ if $f(s) = \bot$, and $-f(s) = \bot$ if $f(s) = \top$. For a set $T \subseteq S$ of states, we write $[T]$: $S \to V$ for the valuation with $[T](s) = \top$ if $s \in T$, and $[T](s) = \bot$ if $s \notin T$. For a proposition $p \in P$, we write $[p]$: $S \to V$ for the valuation with $[p](s) = \top$ if $p \in \langle s \rangle$, and $[p](s) = \bot$ if $p \notin \langle s \rangle$.

Predecessor operators. Let V be a value lattice. Let Pre be a family of functions that contains, for every game structure G, a strict (i.e., bottom-preserving), monotone, and continuous function Pre^G: $[S^G \to V] \to [S^G \to V]$. The function family Pre is a *predecessor-1 operator on* V if for every game structure G, every player-1 restriction G_1 of G, every player-2 restriction G_2 of G, and every valuation f: $S^G \to V$, both $Pre^G(f) \sqsupseteq Pre^{G_1}(f)$ and $Pre^G(f) \sqsubseteq Pre^{G_2}(f)$. Symmetrically, the function family Pre is a *predecessor-2 operator on* V if for every game structure G, every player-1 restriction G_1 of G, every player-2 restriction G_2 of G, and every valuation f: $S^G \to V$, we have both $Pre^G(f) \sqsubseteq Pre^{G_1}(f)$ and $Pre^G(f) \sqsupseteq Pre^{G_2}(f)$. Intuitively, the more actions are available to player 1 in a game structure, the "better" (i.e., closer to top in the valuation lattice) the result of applying a predecessor-1 operator to a valuation, and the "worse" (i.e., closer to bottom) the result of applying a predecessor-2 operator.

Example 1: boolean game structures ("control"). Consider the *boolean value lattice* $V_{\mathbb{B}} = (\mathbb{B}, \vee, \wedge, \mathtt{T}, \mathtt{F})$, where truth $\mathtt{T}$ is the top element and falsehood $\mathtt{F}$ is the bottom element. The valuations for a game structure G on $V_{\mathbb{B}}$ are called the *boolean valuations* for G; they correspond to the subsets of S^G. Figure 1 defines the predecessor operators $1Pre_{\mathbb{B}}$ and $2Pre_{\mathbb{B}}$, applied to a game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$, boolean valuation f: $S \to \mathbb{B}$, and state $s \in S$. For a set $T \subseteq S$ of states, the boolean valuation $1Pre_{\mathbb{B}}^G[T]$: $S \to \mathbb{B}$ of "controllable predecessors" is true at the states from which player 1 can force the game into T in a single step, no matter which action player 2 chooses. The operator $2Pre_{\mathbb{B}}$ behaves symmetrically for player 2, and therefore solves the control problem for the player-2 objective of reaching the target set T in a single step. The operator $1Pre_{\mathbb{B}}$ is a predecessor-1 operator on $V_{\mathbb{B}}$, and $2Pre_{\mathbb{B}}$ is a predecessor-2 operator.

Example 2: boolean transition structures ("verification"). Consider again the boolean value lattice $V_{\mathbb{B}}$. Figure 1 defines the predecessor operators $EPre_{\mathbb{B}}$ and $APre_{\mathbb{B}}$. For a set $T \subseteq S$ of states, the boolean valuation $EPre_{\mathbb{B}}^G[T]$: $S \to \mathbb{B}$ of "possible predecessors" is true at the states that have some successor in T; the boolean valuation $APre_{\mathbb{B}}^G[T]$: $S \to \mathbb{B}$ of "unavoidable predecessors" is true at the states that have all successors in T. For each game structure G, the functions $EPre_{\mathbb{B}}^G$ and $APre_{\mathbb{B}}^G$ correspond to the branching-time "next" operators $\exists\bigcirc$ and $\forall\bigcirc$, respectively, of temporal logic interpreted over the underlying transition structure K^G. Therefore, $EPre_{\mathbb{B}}$ and $APre_{\mathbb{B}}$ solve the verification problems with the specifications of possibly or certainly reaching the target set T in a single step. The operators $EPre_{\mathbb{B}}$ and $APre_{\mathbb{B}}$ are both predecessor-1 and predecessor-2 operators on $V_{\mathbb{B}}$.

Example 3: quantitative game structures ("optimal control"). Consider the *quantitative value lattice* $V_{\mathbb{P}} = (\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, \max, 0, \infty)$, where 0 is the top element and ∞ is the bottom element. Intuitively, each value represents a cost, and the smaller the cost, the "better." In particular, $u \sqsubseteq v$ iff either $u, v \in \mathbb{R}_{\geq 0}$ and $u \geq v$, or $u = \infty$; that is, the lattice

$$\begin{Bmatrix} 1Pre^G_{\mathbb{P}} \\ 2Pre^G_{\mathbb{P}} \end{Bmatrix}(f)(s) = \begin{Bmatrix} \min_{a\in\Gamma_1(s)} \cdot \max_{b\in\Gamma_2(s)} \\ \min_{b\in\Gamma_2(s)} \cdot \max_{a\in\Gamma_1(s)} \end{Bmatrix}. \; w(s,a,b) + f(\delta(s,a,b)) \tag{7}$$

$$\begin{Bmatrix} EPre^G_{\mathbb{P}} \\ APre^G_{\mathbb{P}} \end{Bmatrix}(f)(s) = \begin{Bmatrix} \min_{a\in\Gamma_1(s)} \cdot \min_{b\in\Gamma_2(s)} \\ \max_{a\in\Gamma_1(s)} \cdot \max_{b\in\Gamma_2(s)} \end{Bmatrix}. \; w(s,a,b) + f(\delta(s,a,b)) \tag{8}$$

Figure 2: Quantitative game and transition predecessor operators

is based on the reverse ordering of the reals. The valuations for a game structure G on $V_{\mathbb{P}}$ are called the *quantitative valuations* for G; they are the functions from S^G to the interval $[0,\infty]$. Figure 2 defines the predecessor operators $1Pre_{\mathbb{P}}$ and $2Pre_{\mathbb{P}}$, applied to a game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$, quantitative valuation f: $S \to [0,\infty]$, and state $s \in S$. For a set $T \subseteq S$ of states, the quantitative valuation $1Pre^G_{\mathbb{P}}[T]$: $S \to [0,\infty]$ gives for each state the minimal cost for player 1 of forcing the game into T in a single step (if player 1 cannot force the game into T, then the cost is ∞). The operator $2Pre_{\mathbb{P}}$ behaves symmetrically for player 2, and therefore solves the optimal-control problem with the player-2 objective of reaching the target set T in a single step at minimal cost. The operator $1Pre_{\mathbb{P}}$ is a predecessor-1 operator on $V_{\mathbb{P}}$, and $2Pre_{\mathbb{P}}$ is a predecessor-2 operator.

Example 4: quantitative transition structures ("optimization"). Consider again the quantitative value lattice $V_{\mathbb{P}}$. Figure 2 defines the predecessor operators $EPre_{\mathbb{P}}$ and $APre_{\mathbb{P}}$. For a set $T \subseteq S$ of states, the quantitative valuation $EPre^G_{\mathbb{P}}[T]$: $S \to [0,\infty]$ gives for each state the weight of the minimal transition into T (or ∞, if no such transition exists), and $APre^G_{\mathbb{P}}[T]$: $S \to [0,\infty]$ gives for each state the weight of the maximal transition into T (or ∞, if some transition does not lead into T). These are the single-step shortest-path and single-step longest-path problems on the underlying transition structure K^G. The operators $EPre_{\mathbb{P}}$ and $APre_{\mathbb{P}}$ are both predecessor-1 and predecessor-2 operators on $V_{\mathbb{P}}$.

2.3 Multi-step verification and control

Multi-step verification ("Can a target set be reached in some number of steps?"), optimization ("What is the shortest path to the target?"), and control problems ("Can one player force the game into the target, in some number of steps, no matter what the other player does?") can be solved by iterating the single-step solutions ("dynamic programming"). Here, we exemplify the solutions for the goals of reachability and safety; more general objectives will be dealt with in Section 4. In the following, consider a game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$ and a proposition $p \in P$.

Reachability. We define $\Diamond p$ to be the set of minimal finite runs of G that end in a p-state; that is, the run $s_0(a_0,b_0)s_1(a_1,b_1)\ldots s_m$ is in $\Diamond p$ iff (1) $p \in \langle s_m\rangle$ and (2) for all $0 \le j < m$, we have $p \notin \langle s_j\rangle$. Figure 3 defines four boolean valuations in $[S \to \mathbb{B}]$. The valuation $\langle\langle 1\rangle\rangle^G_{\mathbb{B}}\Diamond p$ is true at the states from which player 1 can control the game to reach a p-state; the valuation $\langle\langle 2\rangle\rangle^G_{\mathbb{B}}\Diamond p$ is true at the states from which player 2 can control the game to reach a p-state; the valuation $\exists^G_{\mathbb{B}}\Diamond p$ is true at the states from which the two players can collaborate to reach a p-state; the valuation $\forall^G_{\mathbb{B}}\Diamond p$ is true at the states from which no matter what the two players do, a p-state will be reached. The first two valuations specify boolean games with the reachability objective $\Diamond p$ for players 1 and 2, respectively; the last two valuations specify the branching-time properties $\exists\Diamond p$ and $\forall\Diamond p$ on the underlying transition structures.

Figure 3 also defines the four corresponding quantitative valuations in $[S \to [0,\infty]]$; we use the convention that the infimum of an empty set of nonnegative reals is ∞, and the supremum is 0. The valuation $\langle\langle 1\rangle\rangle^G_{\mathbb{P}}\Diamond p$ gives for each state the minimal cost for player 1 to direct the game to a p-state (or ∞, if player 1 cannot direct the game to a p-state); the valuation $\langle\langle 2\rangle\rangle^G_{\mathbb{P}}\Diamond p$ gives for each state the minimal cost for player 2 to direct the game to a p-state; the valuation $\exists^G_{\mathbb{P}}\Diamond p$ gives for each state the minimal cost to reach a p-state if both players collaborate; the valuation $\forall^G_{\mathbb{P}}\Diamond p$ gives for each state the maximal reward achievable, if both players collaborate, before a p-state is reached. The first two valuations specify quantitative games with the reachability objective $\Diamond p$ for players 1 and 2, respectively; the last two valuations specify shortest-path and longest-path problems on the underlying transition structure.

The boolean and quantitative valuations for the reachability objective $\Diamond p$ can be characterized by least-fixpoint expressions on the corresponding valu-

$$\begin{Bmatrix} \langle\langle 1\rangle\rangle^G_{\mathbb{B}} \\ \langle\langle 2\rangle\rangle^G_{\mathbb{B}} \end{Bmatrix}(\Diamond p)(s) = \begin{Bmatrix} \exists\xi_1 \in \Xi_1.\forall\xi_2 \in \Xi_2 \\ \exists\xi_2 \in \Xi_2.\forall\xi_1 \in \Xi_1 \end{Bmatrix}. (R_{\xi_1,\xi_2}(s) \text{ has a prefix in } \Diamond p) \quad (9)$$

$$\begin{Bmatrix} \exists^G_{\mathbb{B}} \\ \forall^G_{\mathbb{B}} \end{Bmatrix}(\Diamond p)(s) = \begin{Bmatrix} \exists\xi_1 \in \Xi_1.\exists\xi_2 \in \Xi_2 \\ \forall\xi_2 \in \Xi_2.\forall\xi_1 \in \Xi_1 \end{Bmatrix}. (R_{\xi_1,\xi_2}(s) \text{ has a prefix in } \Diamond p) \quad (10)$$

$$\begin{Bmatrix} \langle\langle 1\rangle\rangle^G_{\mathbb{P}} \\ \langle\langle 2\rangle\rangle^G_{\mathbb{P}} \end{Bmatrix}(\Diamond p)(s) = \begin{Bmatrix} \inf_{\xi_1\in\Xi_1} . \sup_{\xi_2\in\Xi_2} \\ \inf_{\xi_2\in\Xi_2} . \sup_{\xi_1\in\Xi_1} \end{Bmatrix}. \{w(r) \mid r \text{ is a prefix of } R_{\xi_1,\xi_2}(s) \text{ and } r \in \Diamond p\} \quad (11)$$

$$\begin{Bmatrix} \exists^G_{\mathbb{P}} \\ \forall^G_{\mathbb{P}} \end{Bmatrix}(\Diamond p)(s) = \begin{Bmatrix} \inf_{\xi_1\in\Xi_1} . \inf_{\xi_2\in\Xi_2} \\ \sup_{\xi_2\in\Xi_2} . \sup_{\xi_1\in\Xi_1} \end{Bmatrix}. \{w(r) \mid r \text{ is a prefix of } R_{\xi_1,\xi_2}(s) \text{ and } r \in \Diamond p\} \quad (12)$$

Figure 3: Boolean and quantitative reachability games

ation lattice:

$$\langle\langle 1\rangle\rangle^G_V \Diamond p = \mu x.\ ([p] \sqcup 1Pre^G_V(x)), \quad (13)$$
$$\langle\langle 2\rangle\rangle^G_V \Diamond p = \mu x.\ ([p] \sqcup 2Pre^G_V(x)), \quad (14)$$
$$\exists^G_V \Diamond p = \mu x.\ ([p] \sqcup EPre^G_V(x)), \quad (15)$$
$$\forall^G_V \Diamond p = \mu x.\ ([p] \sqcup APre^G_V(x)). \quad (16)$$

where $V \in \{\mathbb{B}, \mathbb{R}\}$, and the variable x ranges over the boolean valuations in $[S \to \mathbb{B}]$ if $V = \mathbb{B}$, and over the quantitative valuations in $[S \to [0, \infty]]$ if $V = \mathbb{R}$. In other words, a single fixpoint expression (namely, "$\Diamond p = \mu x.(p \vee pre(x))$") suffices for the solution of boolean and quantitative verification and control problems, provided the *pre*-operator is interpreted appropriately.

Fixpoint expressions prescribe algorithms. The solutions to the fixpoint equations (13) (16) can be computed iteratively on the valuation lattice as the limit of a sequence $x_0, x_1, x_2, \ldots$ of valuations: let $x_0 = \bot$, and for all $k \geq 0$, let $x_{k+1} = [p] \sqcup Pre^G_V(x_k)$, where $Pre \in \{1Pre, 2Pre, EPre, APre\}$. For our four examples, the iteration converges in a finite number of steps. This is well-known in the case of boolean game structures and in the case of quantitative transition structures; finite convergence can be shown inductively also for quantitative game structures.

Safety. The complement of a reachability objective is a safety objective. We define $\Box p$ to be the set of infinite runs of the game structure G that never leave p-states; that is, the run $s_0(a_0, b_0)s_1(a_1, b_1)\ldots$ is in $\Box p$ iff $p \in \langle s_j \rangle$ for all $j \geq 0$. Figure 4 defines the boolean and quantitative valuations for the safety objective $\Box p$. For example, the boolean valuation $\langle\langle 1\rangle\rangle^G_{\mathbb{B}} \Box p$ is true at the states from which player 1 can control the game to stay within p-states; the quantitative valuation $\exists^G_{\mathbb{P}} \Box p$ gives for each state the minimal cost of an infinite path that stays within p-states; the boolean valuation $\forall^G_{\mathbb{B}} \Box p$ is true at the states from which p is an invariant.

The boolean and quantitative valuations for the safety objective $\Box p$ can be characterized by greatest-fixpoint expressions on the corresponding valuation lattice:

$$\langle\langle 1\rangle\rangle^G_V \Box p = \nu x.\ ([p] \sqcap 1Pre^G_V(x)), \quad (21)$$
$$\langle\langle 2\rangle\rangle^G_V \Box p = \nu x.\ ([p] \sqcap 2Pre^G_V(x)), \quad (22)$$
$$\exists^G_V \Box p = \nu x.\ ([p] \sqcap EPre^G_V(x)), \quad (23)$$
$$\forall^G_V \Box p = \nu x.\ ([p] \sqcap APre^G_V(x)), \quad (24)$$

where $V \in \{\mathbb{B}, \mathbb{R}\}$. The solutions to these fixpoint equations can again be computed iteratively as the limit of a sequence $x_0, x_1, x_2, \ldots$ of valuations: let $x_0 = \top$, and for all $k \geq 0$, let $x_{k+1} = [p] \sqcap Pre^G_V(x_k)$. This iteration converges for boolean game structures in a finite number of steps, but not necessarily for quantitative game or transition structures, where convergence may require ω many steps.

3 An Extremal Model Theorem

For verification problems, fixpoint solutions are known for much richer objectives ("specifications") than reachability and safety, and a fixpoint theory — the μ-calculus — is available for this purpose. In the case of reachability and safety, the fixpoint expressions we provided (namely, $\Diamond p = \mu x.(p \vee pre(x))$ and $\Box p = \nu x.(p \wedge pre(x))$) solve both the verification and control problems. This is not always the case: as we pointed out in the introduction, there are fixpoint expressions that solve a verification problem over transition structures, but do not solve the corresponding control problem over game structures. We now characterize the fixpoint expressions that solve both verifi-

$$\langle\langle 1\rangle\rangle^G_{\mathbb{B}}(\Box p)(s) = \exists \xi_1 \in \Xi_1.\forall \xi_2 \in \Xi_2.\ (R_{\xi_1,\xi_2}(s) \in \Box p) \tag{17}$$

$$\left\{\begin{matrix}\exists^G_{\mathbb{B}}\\ \forall^G_{\mathbb{B}}\end{matrix}\right\}(\Box p)(s) = \left\{\begin{matrix}\exists \xi_1 \in \Xi_1.\exists \xi_2 \in \Xi_2\\ \forall \xi_2 \in \Xi_2.\forall \xi_1 \in \Xi_1\end{matrix}\right\}.\ (R_{\xi_1,\xi_2}(s) \in \Box p) \tag{18}$$

$$\langle\langle 1\rangle\rangle^G_{\mathbb{R}}(\Box p)(s) = \inf_{\xi_1 \in \Xi_1}.\ \sup_{\xi_2 \in \Xi_2}.\ \{w(R_{\xi_1,\xi_2}(s)) \mid R_{\xi_1,\xi_2}(s) \in \Box p\} \tag{19}$$

$$\left\{\begin{matrix}\exists^G_{\mathbb{R}}\\ \forall^G_{\mathbb{R}}\end{matrix}\right\}(\Box p)(s) = \left\{\begin{matrix}\inf_{\xi_1 \in \Xi_1}.\inf_{\xi_2 \in \Xi_2}\\ \sup_{\xi_2 \in \Xi_2}.\sup_{\xi_1 \in \Xi_1}\end{matrix}\right\}.\ \{w(R_{\xi_1,\xi_2}(s)) \mid R_{\xi_1,\xi_2}(s) \in \Box p\} \tag{20}$$

Figure 4: Boolean and quantitative safety games

cation and control problems, provided the predecessor operators are interpreted appropriately.

3.1 Linear temporal logic

Consider a game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$. We express winning objectives for the infinite game played on G by formulas of linear temporal logic (LTL). The *LTL formulas* are generated by the grammar

$$\Psi ::= p \mid \neg\Psi \mid \Psi \vee \Psi \mid \bigcirc\Psi \mid \Psi\,\mathcal{U}\,\Psi,$$

where $p \in P$ is a proposition, $\bigcirc$ is the "next" operator, and $\mathcal{U}$ is the "until" operator. Additional constructs such as $\Diamond\Psi = \mathrm{T}\,\mathcal{U}\Psi$ and $\Box\Psi = \neg\Diamond\neg\Psi$ can be defined in the standard way. A *trace* π: $\omega \to 2^P$ is an infinite sequence of sets of propositions. Every LTL formula Ψ has a truth value on each trace. We write $L(\Psi)$ for the set of traces that satisfy Ψ; a formal definition of $L(\Psi)$ can be found in [9].

Boolean LTL games. Every infinite run $r = s_0(a_0, b_0)s_1(a_1, b_1)s_2\ldots$ of the game structure G induces a trace $\langle r\rangle = \langle s_0\rangle\langle s_1\rangle\langle s_2\rangle\ldots$ Consider a state $s \in S$ and an LTL formula Ψ. We say that *player 1 can control state s for objective Ψ in the game structure G* if player 1 has a strategy $\xi_1 \in \Xi_1$ such that for all strategies $\xi_2 \in \Xi_2$ of player 2, the trace induced by the outcome of the game satisfies the formula Ψ; that is, $\langle R_{\xi_1,\xi_2}(s)\rangle \in L(\Psi)$. A suitable strategy ξ_1 is a *winning player-1 strategy for Ψ from s in G*. We write $\langle\langle 1\rangle\rangle^G_{\mathbb{B}}\Psi$: $S \to \mathbb{B}$ for the boolean valuation that is true at the states which can be controlled by player 1 for Ψ in G; see Figure 5. The player-2 winning valuation $\langle\langle 2\rangle\rangle^G_{\mathbb{B}}\Psi$ is defined symmetrically. Figure 5 also defines the boolean valuation $\exists^G_{\mathbb{B}}\Psi$: $S \to \mathbb{B}$, which is true at the states that satisfy the existential CTL* formula $\exists\Psi$ in the underlying transition structure K^G; and the boolean valuation $\forall^G_{\mathbb{B}}\Psi$: $S \to \mathbb{B}$, which is true at the states that satisfy the universal CTL* formula $\forall\Psi$ in K^G.

Quantitative LTL games. By $\langle\langle 1\rangle\rangle^G_{\mathbb{R}}\Psi$ we wish to denote the minimal cost for player-1 to achieve the objective Ψ. Recall the previous section. In reachability games, we compute the cost of winning as the weight of a finite run that reaches the target, while in safety games, the cost of winning is the weight of an infinite run. This is because upon reaching the target, we know that the reachability objective is satisfied, while a safety objective can be witnessed only by the entire infinite run generated by a game. We generalize this principle to arbitrary LTL formulas by defining the satisfaction index of a trace with respect to an LTL formula. Given a trace $\pi = \pi_0\pi_1\pi_2\ldots$ and a nonnegative integer k, the trace $\pi' = \pi'_0\pi'_1\pi'_2\ldots$ is a *k-variant* of π iff $\pi_j = \pi'_j$ for all $0 \le j \le k$. Let $\Lambda(\pi, k)$ be the set of k-variants of π. For a trace π and an LTL formula Ψ, the *satisfaction index* $\kappa(\pi, \Psi)$ is the smallest integer $k \ge 0$ such that $\Lambda(\pi, k) \subseteq L(\Psi)$ if such a k exists, and $\kappa(\pi, \Psi) = \infty$ otherwise. Intuitively, $\kappa(\pi, \Psi)$ the minimal number of steps after which we can conclude that the trace π satisfies the formula Ψ.

For an infinite run r and a nonnegative integer k, let $r[0..k]$ be the the prefix of r that contains k states. Given an LTL formula Ψ, the quantitative valuation $\langle\langle 1\rangle\rangle^G_{\mathbb{R}}\Psi$: $S \to [0, \infty]$ is formally defined in Figure 5. For each state $s \in S$, we say that $\langle\langle 1\rangle\rangle^G_{\mathbb{R}}\Psi(s)$ is the *player-1 value of the game with objective Ψ at the state s of the game structure G*. A strategy ξ_1 that attains the infimum is an *optimal player-1 strategy for Ψ from s in G*. The player-2 valuation $\langle\langle 2\rangle\rangle^G_{\mathbb{R}}\Psi$ is defined symmetrically. Figure 5 also defines the quantitative valuation $\exists^G_{\mathbb{R}}\Psi$: $S \to [0, \infty]$, which for each state s gives the minimum cost necessary for determining that some path from s in the underlying transition structure K^G satisfies Ψ (or ∞, if no such path exists). Dually, the valuation $\forall^G_{\mathbb{R}}\Psi$: $S \to [0, \infty]$ gives for each state s the maximal reward attainable along some path from s in K^G until Ψ can no longer be violated.

$$\langle\langle 1\rangle\rangle^G_{\mathbb{B}}\Psi(s) = \exists\xi_1 \in \Xi_1.\forall\xi_2 \in \Xi_2.\ (\langle R_{\xi_1,\xi_2}(s)\rangle \in L(\Psi)) \tag{25}$$

$$\begin{Bmatrix}\exists^G_{\mathbb{B}}\\ \forall^G_{\mathbb{B}}\end{Bmatrix}\Psi(s) = \begin{Bmatrix}\exists\xi_1 \in \Xi_1.\exists\xi_2 \in \Xi_2\\ \forall\xi_2 \in \Xi_2.\forall\xi_1 \in \Xi_1\end{Bmatrix}.\ (\langle R_{\xi_1,\xi_2}(s)\rangle \in L(\Psi)) \tag{26}$$

$$\langle\langle 1\rangle\rangle^G_{\mathbb{P}}\Psi(s) = \inf_{\xi_1\in\Xi_1}.\sup_{\xi_2\in\Xi_2}.\ \{w(r[0..\kappa(\langle r\rangle,\Psi)]) \mid r = R_{\xi_1,\xi_2}(s) \text{ and } \langle r\rangle \in L(\Psi)\} \tag{27}$$

$$\begin{Bmatrix}\exists^G_{\mathbb{P}}\\ \forall^G_{\mathbb{P}}\end{Bmatrix}\Psi(s) = \begin{Bmatrix}\inf_{\xi_1\in\Xi_1}.\inf_{\xi_2\in\Xi_2}\\ \sup_{\xi_2\in\Xi_2}.\sup_{\xi_1\in\Xi_1}\end{Bmatrix}.\ \{w(r[0..\kappa(\langle r\rangle,\Psi)]) \mid r = R_{\xi_1,\xi_2}(s) \text{ and } \langle r\rangle \in L(\Psi)\} \tag{28}$$

Figure 5: Boolean and quantitative LTL games

3.2 Fixpoint calculi for games

We define a family of fixpoint logics on game structures. The *fixpoint formulas* are generated by the grammar

$$\varphi ::= p \mid \neg p \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid pre_1(\varphi) \mid pre_2(\varphi) \mid \mu x.\varphi \mid \nu x.\varphi,$$

for propositions $p \in P$ and variables x. A fixpoint formula φ is a *one-player formula* if either it contains no pre_2-operator, or it contains no pre_1-operator. In the former case, φ is a *player-1 formula*; in the latter case, a *player-2 formula*. Given a value lattice V, a predecessor-1 operator Pre_1 on V, and a predecessor-2 operator Pre_2 on V, the closed fixpoint formulas form a logic on game structures: for every game structure G, every closed fixpoint formula $\varphi(Pre_1, Pre_2)$ specifies a valuation $[\![\varphi]\!]^G$: $S^G \to V$. The syntactic operator pre_1 is interpreted semantically as the predecessor-1 operator Pre_1, and pre_2 is interpreted as Pre_2. To make the interpretation of the pre-operators explicit, we sometimes write $\varphi(Pre_1, Pre_2)$ when naming a fixpoint formula. Then, $\varphi(Pre'_1, Pre'_2)$ describes the syntactically identical fixpoint formula, with the pre_1-operator interpreted as Pre'_1, and pre_2 interpreted as Pre'_2. Likewise, the one-player formulas have only a single predecessor operator as argument.

We now define the semantics of fixpoint formulas formally. Let V be a value lattice V, let Pre_1 be a predecessor-1 operator on V, and let Pre_2 be a predecessor-2 operator on V. Let G be a game structure. A *variable environment* $\mathcal{E}$ for G is a function that maps every variable x to a valuation in $[S^G \to V]$. We write $\mathcal{E}[x \mapsto f]$ for the function that agrees with $\mathcal{E}$ on all variables, except that x is mapped to the valuation f. Given V, Pre_1, Pre_2, G, and a variable environment $\mathcal{E}$ for G, each fixpoint formula φ specifies a valuation $[\![\varphi]\!]^G_{\mathcal{E}}$: $S_G \to V$, which is defined inductively by the following equations:

$$\begin{aligned}
[\![p]\!]^G_{\mathcal{E}} &= [p]\\
[\![\neg p]\!]^G_{\mathcal{E}} &= -[p]\\
[\![x]\!]^G_{\mathcal{E}} &= \mathcal{E}(x)\\
[\![\varphi_1 \begin{Bmatrix}\vee\\ \wedge\end{Bmatrix} \varphi_2]\!]^G_{\mathcal{E}} &= [\![\varphi_1]\!]^G_{\mathcal{E}} \begin{Bmatrix}\sqcup\\ \sqcap\end{Bmatrix} [\![\varphi_2]\!]^G_{\mathcal{E}}\\
[\![\begin{Bmatrix}pre_1\\ pre_2\end{Bmatrix}(\varphi)]\!]^G_{\mathcal{E}} &= \begin{Bmatrix}Pre_1^G\\ Pre_2^G\end{Bmatrix}[\![\varphi]\!]^G_{\mathcal{E}}\\
[\![\begin{Bmatrix}\mu\\ \nu\end{Bmatrix}x.\ \varphi]\!]^G_{\mathcal{E}} &= \begin{Bmatrix}\sqcap\\ \sqcup\end{Bmatrix}\{f : S^G \to V \mid f = [\![\varphi]\!]^G_{\mathcal{E}[x\mapsto f]}\}
\end{aligned}$$

All right-hand-side (semantic) operations are performed on the valuation lattice $[S^G \to V]$. If φ is a closed formula, then $[\![\varphi]\!]^G = [\![\varphi]\!]^G_{\mathcal{E}}$ for any variable environment $\mathcal{E}$.

Provided that the predecessor operators Pre_1 and Pre_2 on V are computable, each formula $\varphi(Pre_1, Pre_2)$ prescribes a dynamic program for computing the valuation $[\![\varphi]\!]^G$ over a game structure G by iterative approximation.

Example: mu-calculus. Choose the boolean value lattice $V_{\mathbb{B}}$, and the predecessor operators $Pre_1 = EPre_{\mathbb{B}}$ and $Pre_2 = APre_{\mathbb{B}}$. The resulting logic on game structures coincides is the μ-calculus [8] on the underlying transition structures.

Example: boolean game calculus. Choose the boolean value lattice $V_{\mathbb{B}}$, and the predecessor operators $Pre_1 = 1Pre_{\mathbb{B}}$ and $Pre_2 = 2Pre_{\mathbb{B}}$. The resulting logic on game structures is the alternating-time μ-calculus of [1]. The player-i fragment, for $i = 1, 2$, is expressive enough to compute the winning states for player i with respect to any LTL objective.

Example: quantitative game calculus. Choose the quantitative value lattice $V_{\mathbb{P}}$, and the predecessor operators $Pre_1 = 1Pre_{\mathbb{P}}$ and $Pre_2 = 2Pre_{\mathbb{P}}$. The resulting logic may be called the *quantitative game calculus*. We shall see that the player-i fragment, for $i = 1, 2$, is expressive enough to compute all player-i values with respect to any LTL objective.

Example: quantitative mu-calculus. Choose the quantitative value lattice $V_{\mathbb{P}}$, and the predecessor op-

erators $Pre_1 = EPre_{\mathbb{R}}$ and $Pre_2 = APre_{\mathbb{R}}$. The resulting logic may be called the *quantitative μ-calculus*. It can be used to compute, for example, the minimal and maximal weights of paths that satisfy LTL formulas in transition structures.

Monotonicity. The following monotonicity property of fixpoint formulas will be useful.

Lemma 1 *For every game structure G, every 1-restriction G_1 of G, every 2-restriction G_2 of G, and every player-1 fixpoint formula φ, we have $[\![\varphi]\!]^G \sqsupseteq [\![\varphi]\!]^{G_1}$ and $[\![\varphi]\!]^G \sqsubseteq [\![\varphi]\!]^{G_2}$. A symmetrical result holds for player-2 formulas.*

Lean fixpoint formulas. We shall use fixpoint formulas as algorithms for computing the values of LTL games. The quantitative interpretation of a fixpoint formula, however, does not take into account the satisfaction index of the corresponding LTL formula, and may compute the cost of a trace even beyond the satisfaction index. For example, the LTL formula $\bigcirc \mathrm{T}$ has the satisfaction index 0, because every state has a successor. Hence $(\exists^G_{\mathbb{P}} \bigcirc \mathrm{T})(s) = 0$ for all game structures G and states $s \in S^G$. While $\exists^G_{\mathbb{R}} \bigcirc \mathrm{T} = [\![EPre_{\mathbb{R}}(\mathrm{T})]\!]^G$ for all game structures G, if $s \in S^G$ is a state all of whose outgoing transitions have positive weights, then $[\![EPre_{\mathbb{P}}(\mathrm{T})]\!]^G(s) > 0$. This motivates the definition of lean fixpoint formulas. A fixpoint formula φ is *valid* if for every game structure G and every state $s \in S^G$, we have $[\![\varphi(1Pre_{\mathbb{R}}, 2Pre_{\mathbb{R}})]\!]^G(s) = \mathrm{T}$. A fixpoint formula is *lean* if no valid subformula contains *pre*-operators.

From now on we will make heavy use of the following convenient notation. If f^G and g^G are two families of valuations, one each for every game structure G, then we write $f = g$ short for "$f^G = g^G$ for all game structures G."

Lemma 2 *Let Ψ be an LTL formula, and let φ be a lean one-player fixpoint formula. Then $\exists_{\mathbb{R}}\Psi = [\![\varphi(EPre_{\mathbb{R}})]\!]$ iff $\exists_{\mathbb{P}}\Psi = [\![\varphi(EPre_{\mathbb{P}})]\!]$, and $\forall_{\mathbb{R}}\Psi = [\![\varphi(APre_{\mathbb{R}})]\!]$ iff $\forall_{\mathbb{P}}\Psi = [\![\varphi(APre_{\mathbb{P}})]\!]$.*

3.3 From verification to control: a semantic criterion

The following theorem characterizes the fixpoint formulas that can be used for solving boolean as well as quantitative games with LTL winning objectives. The characterization reduces problems on two-player structures (control) and on quantitative structures (optimization) to problems on *boolean one-player* structures (verification), which are well-understood.

Theorem 1 *For every LTL formula Ψ and every lean player-i fixpoint formula φ, where $i = 1, 2$, the following four statements are equivalent:*

- $\langle\langle i \rangle\rangle_{\mathbb{R}}\Psi = [\![\varphi(iPre_{\mathbb{R}})]\!]$.
- $\langle\langle i \rangle\rangle_{\mathbb{B}}\Psi = [\![\varphi(iPre_{\mathbb{B}})]\!]$.
- $\exists_{\mathbb{R}}\Psi = [\![\varphi(EPre_{\mathbb{R}})]\!]$ *and* $\forall_{\mathbb{R}}\Psi = [\![\varphi(APre_{\mathbb{R}})]\!]$.
- $\exists_{\mathbb{B}}\Psi = [\![\varphi(EPre_{\mathbb{B}})]\!]$ *and* $\forall_{\mathbb{B}}\Psi = [\![\varphi(APre_{\mathbb{B}})]\!]$.

The theorem can be stated equivalently as follows:

> $\langle\langle i \rangle\rangle^G_{\mathbb{P}}\Psi = [\![\varphi(iPre_{\mathbb{R}})]\!]^G$ for all game structures G iff $\langle\langle i \rangle\rangle^G_{\mathbb{B}}\Psi = [\![\varphi(iPre_{\mathbb{B}})]\!]^G$ for all one-player structures G.

In other words, the fixpoint formula φ prescribes an algorithm for computing the boolean or quantitative values of games with the winning objective Ψ iff it does so on all boolean, extremal game structures, where one or the other player has no choice of actions.

Proof sketch. Clearly, a fixpoint formula φ that solves games with objective Ψ also works over one-player structures, which are special cases of games. For the implication from one-player to game structures, we argue by contradiction. We start with the boolean player-1 interpretation (the proof for player 2 is symmetric). First we notice that given a game structure G for which the two valuations $\langle\langle 1 \rangle\rangle^G_{\mathbb{B}}\Psi$ and $[\![\varphi(1Pre_{\mathbb{R}})]\!]^G$ differ, we can construct a turn-based game structure G' for which the valuations differ as well. There are two cases. If $\langle\langle 1 \rangle\rangle^{G'}_{\mathbb{B}}\Psi(s) < [\![\varphi(1Pre_{\mathbb{R}})]\!]^{G'}(s)$ for some state $s \in S^{G'}$, then we fix a finite-memory optimal strategy of player 2 and show that in the resulting player-1 structure G_1, there is a state t such that $(\exists^{G_1}_{\mathbb{B}}\Psi)(t) < [\![\varphi(EPre_{\mathbb{R}})]\!]^{G_1}(t)$. Similarly, if $\langle\langle 1 \rangle\rangle^{G'}_{\mathbb{B}}\Psi(s) > [\![\varphi(1Pre_{\mathbb{R}})]\!]^{G'}(s)$ for some state $s \in S^{G'}$, then we fix a finite-memory optimal strategy of player 1 and argue on the resulting player-2 structure. The proof for quantitative games follows by a similar argument. Finally, we go from quantitative to boolean structures using Lemma 2. □

Suppose we are given an LTL formula Ψ. For verifying whether some path of a transition structure K^G satisfies Ψ, we can construct a μ-calculus formula $\varphi(EPre_{\mathbb{R}})$ that is equivalent to $\exists_{\mathbb{R}}\Psi$ over all transition structures, and check $\varphi(EPre_{\mathbb{R}})$ over K^G; this is, in fact, a symbolic model checking algorithm for LTL [3]. Now suppose that we want player 1 to control the game structure G for the objective Ψ. Theorem 1 tells us whether we can simply substitute the controllable predecessor operator $1Pre_{\mathbb{R}}$ for the μ-calculus predecessor operator $EPre_{\mathbb{R}}$ in the fixpoint formula φ: the substitution works if and only if by substituting $APre_{\mathbb{R}}$

for $EPre_\mathbb{R}$ in φ we obtain a formula that is equivalent to the universal interpretation $\forall_\mathbb{R}\Psi$ of the LTL formula over all transition structures.

To see that this property is not trivial (i.e., not satisfied by every μ-calculus formula $\varphi(EPre_\exists)$ that is equivalent to $\exists_\mathbb{R}\Psi$), consider the co-Büchi formula $\Psi = \Diamond\Box p$. Over transition structures, $\exists\Diamond\Box p$ is equivalent to $\exists\Diamond\exists\Box p$, which is equivalent to the μ-calculus formula $\mu x.(\nu y.(p \wedge EPre_\mathbb{R}(y)) \vee EPre_\mathbb{R}(x))$; indeed, this is the result of the standard translation from LTL to the μ-calculus for co-Büchi formulas [7, 4]. However, the corresponding game formula $\mu x.(\nu y.(p \wedge 1Pre_\mathbb{R}(y)) \vee 1Pre_\mathbb{R}(x))$ does not compute the boolean valuation $\langle\langle 1\rangle\rangle^G_\mathbb{R}\Diamond\Box p$ for all game structures G: the game structure given in the introduction provides a counterexample. The criterion of Theorem 1 fails, because over transition structures, $\forall_\exists\Diamond\Box p$ is not equivalent to $\forall\Diamond\forall\Box p$, and therefore $\forall_\exists\Psi$ is not equivalent to $\mu x.(\nu y.(p \wedge APre_\mathbb{R}(y)) \vee APre_\exists(x))$. This is not surprising, given that the solution of ω-regular games requires deterministic (and hence Rabin chain) ω-automata [15], whereas nondeterministic (and hence Büchi) ω-automata suffice for ω-regular verification. The translations of [7, 4] from LTL to the μ-calculus go via nondeterministic Büchi automata, and thus cannot be used to solve ω-regular games.

The following theorem characterizes the cost of checking the criterion given in Theorem 1. There is a gap between the lower and upper bounds, which is due to the gap between the best known lower and upper bounds for the equivalence problem between an LTL formula and a μ-calculus formula.

Theorem 2 *Let Ψ be an LTL formula, and let φ be a one-player fixpoint formula. The complexity of checking whether $\exists_\mathbb{R}\Psi = [\![\varphi]\!]$ is in 2EXPTIME and PSPACE-hard in the size of Ψ, and in EXPTIME in the size of φ. The complexity of checking whether $\forall_\mathbb{R}\Psi = [\![\varphi]\!]$ is the same.*

3.4 From verification to control: a syntactic criterion

Not all fixpoint formulas correspond to verification or control problems with respect to linear-time objectives. This is always the case, however, for the deterministic fixpoint formulas. The *deterministic* fixpoint formulas are generated by the grammar

$$\varphi ::= p \mid \neg p \mid x \mid \varphi \vee \varphi \mid p \wedge \varphi \mid \neg p \wedge \varphi \mid pre_1(\varphi) \mid pre_2(\varphi) \mid \mu x.\varphi \mid \nu x.\varphi.$$

From [6] we know that if $\varphi(EPre_\mathbb{R})$ is a one-player deterministic fixpoint formula, then there is an ω-regular language Θ such that $\exists_\mathbb{R}\Theta = [\![\varphi(EPre_\mathbb{R})]\!]$. However, the examples (2) and (3) in the introduction illustrate that for such a formula $\varphi(EPre_\mathbb{R})$, in general it is not the case that $\langle\langle 1\rangle\rangle_\mathbb{R}\Theta = [\![\varphi(1Pre_\mathbb{R})]\!]$. In other words, the correspondence between the deterministic fixpoint formula and the ω-regular language does not necessarily carry over from verification to control. It is then natural to ask what other conditions we need, in addition to determinism, for a one-player fixpoint formula to have related meanings in verification and control. We answer this question by introducing a subclass of the deterministic formulas. A fixpoint formula φ is *strongly deterministic* iff φ consists of a string of fixpoint quantifiers followed by a quantifier-free part ψ, which is generated by the grammar

$$\begin{aligned}\psi &::= p \mid \neg p \mid \psi \vee \psi \mid p \wedge \psi \mid \neg p \wedge \psi \mid pre_1(\chi) \mid pre_2(\chi),\\ \chi &::= x \mid \chi \vee \chi.\end{aligned}$$

Note that every strongly deterministic fixpoint formula is lean. The following theorem shows that the one-player strongly deterministic fixpoint formulas provide a syntactic class of fixpoint formulas for which the criterion of Theorem 1 applies. In particular, it follows that for every LTL formula Ψ, every one-player strongly deterministic fixpoint formula φ, and $i = 1, 2$, we have $\langle\langle i\rangle\rangle.\Psi = [\![\varphi(iPre_\cdot)]\!]$.

Theorem 3 *For every LTL formula Ψ and every one-player strongly deterministic fixpoint formula φ, we have $\exists_\cdot\Psi = [\![\varphi(EPre_\cdot)]\!]$ iff $\forall_\cdot\Psi = [\![\varphi(APre_\cdot)]\!]$.*

Proof sketch. A strongly deterministic formula starts with a quantifier prefix. In the sequence $\mu x_1.\nu x_2 \ldots \nu x_{2k}$ of alternating fixpoints, the "evaluation order" is $x_2 \succ x_4 \succ \cdots \succ x_{2k} \succ x_{2k-1} \succ \cdots \succ x_1$ (this reflects the extension of the variables when the expression is being evaluated). Using this evaluation order, every one-player strongly deterministic fixpoint formula $\varphi(EPre_\cdot)$ can be brought into the normal form $\mu x_1.\nu x_2 \ldots \nu x_{2k}.(d_0 \vee \bigvee_{j=1}^{2k}(d_j \wedge EPre_\mathbb{R}(x_j)))$, for some $k > 0$ and some mutually exclusive boolean combinations $d_0, d_1, \ldots, d_{2k}$ of propositions. The theorem follows from the fact that this formula has essentially the same structure as the solution formula of a Rabin-chain game (cf. [5] and Section 4). □

While the one-player strongly deterministic fixpoint formulas obey strict syntactic conditions, the proof of Theorem 3 shows that they suffice for solving all control problems with Rabin-chain objectives. In turn, every ω-regular property can be specified by a deterministic Rabin-chain automaton [10, 15]. We can therefore transform every control problem with an ω-regular objective into a control problem with a Rabin-chain objective that is to be solved on the automata-

theoretic product of the given game structure and a Rabin-chain automaton. Hence, at the cost of possibly enlarging the game structure, the one-player strongly deterministic fixpoint formulas suffice for the solution of games with arbitrary ω-regular objectives.

4 Dynamic Programs for LTL

We show that for every LTL formula Ψ we can construct an equivalent fixpoint formula φ_Ψ that meets the criterion of Theorem 1. The formula φ_Ψ has the following properties: it solves both the verification problem (on transition structures) for specification Ψ and the control problem (on game structures) for objective Ψ, both under boolean and quantitative interpretations. The construction of φ_Ψ is optimal for the boolean case, in that the 2EXPTIME complexity of the resulting algorithm for solving boolean games with LTL objectives matches the hardness of the problem [11].

4.1 (Co)Büchi and Rabin-chain games

The objective of a *Büchi game* is an LTL formula of the form $\Box\Diamond p$, for a proposition $p \in P$, and the objective of a *co-Büchi game* is an LTL formula of the form $\Diamond\Box p$. For $V = \{\mathbb{B}, \mathbb{R}\}$ and $i = 1, 2$, the Büchi and co-Büchi valuations can be computed by the fixpoint formulas

$$\begin{aligned} \langle\langle i\rangle\rangle_V \Box\Diamond p &= [\![\nu y.\mu x.(iPre_V(x) \vee (p \wedge iPre_V(y)))]\!], \\ \langle\langle i\rangle\rangle_V \Diamond\Box p &= [\![\mu x.\nu y.(iPre_V(x) \vee (p \wedge iPre_V(y)))]\!]. \end{aligned}$$

The objective a *Rabin-chain game* is an LTL formula of the form $\Phi = \bigvee_{j=0}^{k-1}(\Box\Diamond d_{2j} \wedge \neg\Box\Diamond d_{2j+1})$, where $k > 0$ is called the *index* of Φ, and $d_0, \ldots, d_{2k}$ are boolean combinations of propositions such that $\emptyset = [d_{2k}] \subseteq [d_{2k-1}] \subseteq \cdots \subseteq [d_0] = S^G$ for all game structures G. An alternative characterization of Rabin-chain games with objective Φ can be obtained by defining a family Ω_Φ^G: $S^G \to \{0, 1, \ldots, 2k-1\}$ of index functions, one for every game structure G, such that $\Omega_\Phi^G(s) = j$ for all states $s \in [d_j] \setminus [d_{j+1}]$. Given an infinite run r of G, let $Inf(r) \subseteq S^G$ be the set of states that occur infinitely often along r, and let $MaxIndex(\Omega_\Phi, r) = \max\{\Omega_\Phi^G(s) \mid s \in Inf(r)\}$ be the largest index of such a state. Then, the run r satisfies the objective Φ iff $MaxIndex(\Omega_\Phi, r)$ is even. For $V \in \mathbb{B}, \mathbb{R}$ and $i = 1, 2$, the Rabin-chain valuation can be computed by the fixpoint formula

$$\begin{aligned} \langle\langle i\rangle\rangle_V \Phi = [\![&\lambda_{2k-1}x_{2k-1}\ldots\mu x_1.\nu x_0. \\ &\textstyle\bigvee_{j=0}^{2k-1}(d_j \wedge \neg d_{j+1} \wedge iPre_V(x_j))]\!], \end{aligned}$$

where $\lambda_j = \nu$ if j is even, and $\lambda_j = \mu$ if j is odd (cf. [5]). Note that the fixpoint solutions for Büchi, co-Büchi, and Rabin-chain games are all one-player strongly deterministic fixpoint formulas.

4.2 LTL games

Given an LTL formula Ψ, we construct a lean one-player fixpoint formula φ_Ψ such that

$$\langle\langle i\rangle\rangle_V \Psi = [\![\varphi_\Psi(iPre_V)]\!] \tag{29}$$

for $V \in \{\mathbb{B}, \mathbb{R}\}$ and $i = 1, 2$. Following [5, 10], our construction is based on deterministic Rabin-chain automata (also called *parity automata* [14]). A *Rabin-chain automaton of index k* over the input alphabet 2^P is a tuple $\mathcal{C} = (Q, Q_0, \Delta, \langle\cdot\rangle, \Omega)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, Δ: $Q \to 2^Q$ is the transition relation, $\langle\cdot\rangle$: $Q \to 2^P$ assigns propositions to states, and Ω: $Q \to \{0, \ldots, 2k-1\}$ is the acceptance condition. An *execution* of $\mathcal{C}$ from a source state $q_0 \in Q$ is an infinite sequence $q_0 q_1 q_2 \cdots$ of automaton states such that $q_{j+1} \in \Delta(q_j)$ for all $j \geq 0$; if $q_0 \in Q_0$, we say that the execution is *initialized*. The execution $e = q_0 q_1 q_2 \ldots$ is *generated* by the trace $\langle e\rangle = \langle q_0\rangle\langle q_1\rangle\langle q_2\rangle \cdots$. The execution e is *accepting* if $MaxIndex(\Omega, e)$ is even. The *language* $L(\mathcal{C})$ is the set of traces π such that $\mathcal{C}$ has an initialized accepting execution e generated by π. The automaton $\mathcal{C}$ is *deterministic and total* if (1a) for all states $q', q'' \in Q_0$, if $q' \neq q''$, then $\langle q'\rangle \neq \langle q''\rangle$; (1b) for all proposition sets $P' \subseteq P$, there is a state $q' \in Q_0$ such that $\langle q'\rangle = P'$; (2a) for all states $q \in Q$ and $q', q'' \in \Delta(q)$, if $q' \neq q''$, then $\langle q'\rangle \neq \langle q''\rangle$; (2b) for all states $q \in Q$ and all proposition sets $P' \subseteq P$, there is a state $q' \in \Delta(q)$ such that $\langle q'\rangle = P'$. If $\mathcal{C}$ is deterministic and total, then we write $\Delta(q, P')$ for the unique state $q' \in \Delta(q)$ with $\langle q'\rangle = P'$.

From the LTL formula Ψ, we construct a deterministic, total Rabin-chain automaton $\mathcal{C}_\Psi$ such that $L(\Psi) = L(\mathcal{C}_\Psi)$, by first building a nondeterministic Büchi automaton with the language $L(\Psi)$ [16], and then determinizing it [12, 13]. Let $\mathcal{C}_\Psi = (Q, Q_0, \Delta, \langle\cdot\rangle, \Omega)$. In order to obtain a lean fixpoint formula φ_Ψ, we need to compute the set $F \subseteq Q$ of automaton states q such that all executions with source q are accepting. To this end, it suffices to compute the set $Q \setminus F$ of states q' such that there is an execution e with source q' and $MaxIndex(\Omega', e)$ is even, where Ω' is the complementary acceptance condition with $\Omega'(q) = (2k-1) - \Omega(q)$ for all states $q \in Q$. This corresponds to checking the nonemptiness of a Rabin-chain automaton [5].

We derive the fixpoint formula φ_Ψ that satisfies (29) in two steps. First, we build a fixpoint for-

mula φ' that solves the game with objective Ψ on the product structure $G \times \mathcal{C}$, for all game structures G. From φ', we then construct the formula φ_Ψ that solves the game directly on G, for all G. Consider an arbitrary game structure $G = (S, \Gamma_1, \Gamma_2, \delta, \langle\cdot\rangle)$. Define $G \times \mathcal{C} = (S', \Gamma_1', \Gamma_2', \delta', \langle\cdot\rangle)$, where $S' = \{(s,q) \in S \times Q \mid \langle s\rangle = \langle q\rangle\}$, where $\Gamma_i'(s,q) = \Gamma_i(s)$ for $i = 1,2$, where $\delta'((s,q), a_1, a_2) = (\delta(s, a_1, a_2), \Delta(q, \langle\delta(s, a_1, a_2)\rangle))$. Finally, for $q \notin F$ let $\langle s,q\rangle = \langle s\rangle \cup \{c_{\Omega(q)}\}$, and for $q \in F$ let $\langle s,q\rangle = \langle s\rangle \cup \{f, c_{\Omega(q)}\}$, where $f, c_0, \ldots, c_{2k-1}$ are new propositions.

We construct φ' by proceeding similarly to [2]. We give the fixpoint formula φ' in equational form; it can then be unfolded into a nested fixpoint formula in the standard way. The formula φ' is composed of blocks $B_0', \ldots, B_{2k-1}'$, where B_0' is the innermost block and B_{2k-1}' the outermost block. The block B_0' is a ν-block which consists of the single equation $x_0 = f \vee \bigvee_{j=0}^{2k-1}(c_j \wedge pre_1(x_j)))$. For $0 < \ell \leq 2k-1$, the block B_ℓ' is a μ-block if ℓ is odd, and a ν-block if ℓ is even; in either case it consists of the single equation $x_\ell = x_{\ell-1}$. The output variable is x_{2k-1}. Then, $(\langle\!\langle 1\rangle\!\rangle_{\mathbb{R}}^G \Psi)(s) = [\![\varphi'(1Pre_{\mathbb{R}})]\!]^{G\times\mathcal{C}}(s,q)$ for all states $s \in S$ and for the unique $q \in Q_0$ such that $(s,q) \in S'$.

The formula φ_Ψ mimics on G the evaluation of φ' on $G \times \mathcal{C}$. For each variable x_ℓ of φ', for $0 \leq \ell \leq 2k-1$, the formula φ_Ψ contains the set $\{x_\ell^q \mid q \in Q\}$ of variables: the value of x_ℓ^q at s keeps track of the value of x_ℓ at (s,q). The formula φ_Ψ is composed of the blocks $B_0, \ldots, B_{2k-1}$: for $0 \leq \ell \leq 2k-1$, the block B_ℓ consists of the set $\{E_\ell^q \mid q \in Q\}$ of equations. The equation E_ℓ^q is derived from the equation for x_ℓ in φ' by replacing the variable x_ℓ on the left-hand side with the variable x_ℓ^q, by replacing c_j with T if $\Omega(q) = j$ and F otherwise, by replacing f with T if $q \in F$ and F otherwise, and by replacing $pre_1(x_j)$ with $pre_1(\bigvee_{q' \in \Delta(q)} x_j^{q'})$; the right-hand side is then conjuncted with the propositions in $\langle q\rangle$. The block B_{2k-1} contains the extra equation $x_{out} = \bigvee_{q \in Q_0} x_{2k-1}^q$, which defines the output variable x_{out}. Note that φ_Ψ is independent of the game structure G, and contains no propositions other than those in Ψ.

Theorem 4 *For every LTL formula Ψ and $i = 1,2$, we have $\langle\!\langle i\rangle\!\rangle_{\mathbb{R}} \Psi = [\![\varphi_\Psi(iPre_{\mathbb{R}})]\!]$. Moreover, the fixpoint formula φ_Ψ is lean and its size is doubly exponential in the size of Ψ.*

Since φ_Ψ is lean, by Theorem 1 it follows that $\langle\!\langle i\rangle\!\rangle_{\mathbb{P}} \Psi = [\![\varphi_\Psi(iPre_{\mathbb{P}})]\!]$. The doubly exponential size of φ_Ψ is optimal, because boolean games with LTL objectives are 2EXPTIME-hard [11].

References

[1] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th Symp. Foundations of Computer Science*, pp. 100–109. IEEE Computer Society, 1997.

[2] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proc. 11th Symp. Logic in Computer Science*, pp. 304–312. IEEE Computer Society, 1996.

[3] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pp. 124–175. Springer, 1994.

[4] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[5] E.A. Emerson and C. Jutla. Tree automata, mu-calculus, and determinacy. In *Proc. 32th Symp. Foundations of Computer Science*, pp. 368–377. IEEE Computer Society, 1991.

[6] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model checking for fragments of μ-calculus. In *CAV 93: Computer-aided Verification*, LNCS 697, pp. 385–396. Springer, 1993.

[7] E.A. Emerson and C. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. First Symp. Logic in Computer Science*, pp. 267–278. IEEE Computer Society, 1986.

[8] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[10] A.W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *Proc. 5th Symp. Computation Theory*, LNCS 208, pp. 157–168. Springer, 1984.

[11] R. Rosner. *Modular Synthesis of Reactive Systems.* PhD Thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[12] S. Safra. On the complexity of ω-automata. In *Proc. 29th Symp. Foundations of Computer Science*, pp. 319–327. IEEE Computer Society, 1988.

[13] S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *Proc. 24th Symp. Theory of Computing*, pp. 275–282. ACM, 1992.

[14] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, vol. B, pp. 133–191. Elsevier, 1990.

[15] W. Thomas. On the synthesis of strategies in infinite games. In *STACS 95: Theoretical Aspects of Computer Science*, LNCS 900, pp. 1–13. Springer, 1995.

[16] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

Deterministic Generators and Games for LTL Fragments *

Rajeev Alur
University of Pennsylvania &
Bell Labs
alur@cis.upenn.edu

Salvatore La Torre †
University of Pennsylvania &
Università degli Studi di Salerno
latorre@seas.upenn.edu

Abstract

Deciding infinite two-player games on finite graphs with the winning condition specified by a linear temporal logic (LTL) formula, is known to be 2EXPTIME-complete. In this paper, we identify LTL fragments of lower complexity. Solving LTL games typically involves a doubly-exponential translation from LTL formulas to *deterministic* ω-automata. First, we show that the *longest distance* (length of the longest simple path) of the generator is also an important parameter, by giving an $O(d \log\ n)$-space procedure to solve a Büchi game on a graph with n vertices and longest distance d. Then, for the LTL fragment with only eventualities and conjunctions, we provide a translation to deterministic generators of exponential size and linear longest distance, show both of these bounds to be optimal, and prove the corresponding games to be PSPACE-complete. Introducing *next* modalities in this fragment, we provide a translation to deterministic generators still of exponential size but also with exponential longest distance, show both of these bounds to be optimal, and prove the corresponding games to be EXPTIME-complete. For the fragment resulting by further adding disjunctions, we provide a translation to deterministic generators of doubly-exponential size and exponential longest distance, show both of these bounds to be optimal, and prove the corresponding games to be EXPSPACE. Finally, we show tightness of the double-exponential bound on the size as well as the longest distance for deterministic generators for LTL even in the absence of *next* and *until* modalities.

*This research was partially supported by NSF Career award CCR97-34115, NSF award CCR99-70925, SRC award 99-TJ-688, and Alfred P. Sloan Faculty Fellowship.

†Partially supported by the M.U.R.S.T. in the framework of project TOSCA.

1 Introduction

Linear temporal logic (LTL) is a popular choice for specifying correctness requirements of reactive systems [14, 13]. An LTL formula is built from state predicates, boolean connectives, and temporal modalities such as *next*, *eventually*, *always*, and *until*, and is interpreted over infinite sequences of states modeling computations of reactive programs. The most studied decision problem concerning LTL is *model checking*: given a finite-state abstraction G of a reactive system and an LTL formula φ, do all infinite computations of G satisfy φ? The first step of the standard solution to model checking involves translating a given LTL formula to a (non-deterministic) Büchi automaton that accepts all of its satisfying models [12, 21]. Such a translation is central to solving the satisfiability problem for LTL also. The translation can be exponential in the worst case, and in fact, both model checking and satisfiability are PSPACE-complete [18].

The standard interpretation of LTL over infinite computations is the natural one for closed systems, where a *closed system* is a system whose behavior is completely determined by the state of the system. However, the compositional modeling and design of reactive systems requires each component to be viewed as an open system, where an *open system* is a system that interacts with its environment and whose behavior depends on the state of the system as well as the behavior of the environment. In the setting of open systems, the key decision problem is to compute the winning strategies in infinite two-player games. In the satisfiability game, we are given an LTL formula φ and a partitioning of atomic propositions into inputs and outputs, and we wish to determine if there is a strategy to produce outputs so that no matter which inputs are supplied, the resulting computation satisfies φ. This problem has been formulated in different contexts such as *synthesis* of reactive modules [15], *realizability* of liveness specifications [4], and *receptiveness* [5]. In the model-checking game, we are given an

0-7695-1281-X/01 $10.00 © 2001 IEEE

LTL specification φ, and a game graph G whose states are partitioned into system states and environment states. We wish to determine if the protagonist has a strategy to ensure that the resulting computation satisfies φ in the infinite game in which the protagonist chooses the successor in all system states and the adversary chooses the successor state in all environment states. This problem appears in contexts such as *module checking* and its variants [9, 10], and the definition of *alternating temporal logic* [2]. Such game-based model checking for restricted formulas such as "always p" has already been implemented in the software MOCHA [3], and shown to be useful in construction of the most-general environments for automating assume-guarantee reasoning [1].

We focus on the game version of model checking: given a game graph G and an LTL formula φ, what is the complexity of deciding whether a given player has a winning strategy starting from a given initial state (game version of satisfaction is a special case, and similar bounds apply). It is known that the complexity of this problem is doubly-exponential in the size of the LTL formula, and the problem is 2EXPTIME-complete [15]. Note that the complexity is much lower for formulas of specific form: generalized Büchi games (formulas of the form $\wedge_i \Box\Diamond p_i$) are solvable in polynomial time, and Streett games (formulas of the form $\wedge_i(\Box\Diamond p_i \rightarrow \Box\Diamond q_i)$) are coNP-complete (the dual, Rabin games are NP-complete) [16, 7]. It is worth mentioning that, in the standard model checking, while full LTL is PSPACE-complete, the fragment which allows only *eventually* and *always* operators (but no *next* or *until*) has a small model property and is NP-complete [18] (see also [6] for complexity results on simpler fragments of LTL). This motivated us to consider the problem addressed in this paper: are there fragments of LTL for which games have complexity lower than 2EXPTIME?

The standard approach to solving games for LTL is by reduction to a game on the product of the game graph and a *deterministic* automaton that accepts all the models of the given formula. The winning condition in this reduced game corresponds to the type of the acceptance condition (e.g. Büchi or Rabin) for the deterministic generator [1]. To obtain a deterministic generator, the standard approach is to first build a nondeterministic generator and then determinize it. Each of these steps costs an exponential, and it is known that there are LTL formulas whose deterministic generators have to be doubly-exponential [11].

In this paper, we give a comprehensive study of deterministic generators and game complexities of various LTL fragments. We use the notation $\text{LTL}(op_1, \ldots, op_k)$ to denote the fragment of LTL given by top-level boolean combination of formulas which use only the boolean connectives and the temporal operators in the list $op_1, \ldots, op_k$. Our first result is a construction of a singly-exponential deterministic Büchi automaton for the fragment $\text{LTL}(\Diamond, \wedge)$. This construction is different from the standard tableau-based construction, and builds the automaton for a formula in a modular way from the automata for its subformulas. This immediately gives a single exponential bound for $\text{LTL}(\Diamond, \wedge)$ games by using the standard algorithm for Büchi games. However, the deterministic generators have the property that the longest simple path is at most linear in the size of the formula. We show that this property can be exploited to reduce space requirement. In fact, we show a general result: in a game graph with n vertices and *longest distance* d (that is, length of longest simple path), a Büchi game can be solved in space $O(d \log\ n)$ (the conventional algorithm uses $O(n)$ space). This leads us to the result that $\text{LTL}(\Diamond, \wedge)$ games can be solved in PSPACE, and we show a matching lower bound. Note that the fragment $\text{LTL}(\Diamond, \wedge)$ contains boolean combinations of invariant ("always p") and termination ("eventually q") properties, and thus includes many of the commonly used specifications.

Combining *next* modalities with the eventualities raises the complexity. For any formula in $\text{LTL}(\Diamond, \bigcirc, \wedge)$, we show how to construct a deterministic Büchi generator with both states and longest distance of exponential size. The construction is optimal since there exists an $\text{LTL}(\Diamond, \bigcirc, \wedge)$ formula for which all deterministic generators must have exponential longest distance. This construction leads to an EXPTIME algorithm for solving games in $\text{LTL}(\Diamond, \bigcirc, \wedge)$, and we show a matching lower bound.

Adding disjunctions to $\text{LTL}(\Diamond, \bigcirc, \wedge)$ raises complexity. Given an $\text{LTL}(\Diamond, \bigcirc, \wedge, \vee)$ formula, we show how to construct a corresponding deterministic Büchi automaton with doubly-exponential states and singly-exponential longest distance. The construction is optimal since we show that there is an $\text{LTL}(\Diamond, \wedge, \vee)$ formula whose deterministic generator must be doubly-exponential with singly-exponential longest distance. Our construction leads to an EXPSPACE algorithm for

[1] In the automata-theoretic formulation of the problem [20], the game graph can be viewed as a tree automaton that generates all the strategies of one of the players. From the formula φ, we can construct a tree automaton that accepts precisely those trees all of whose paths satisfy φ, take product with the game tree automaton, and test for emptiness. This approach has the same computational essence, and requires determinization.

solving games in LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$). A matching lower bound remains an open problem.

The nesting of eventually and always modalities causes a further increase in the complexity. We prove that there exists a formula in LTL($\Box, \Diamond, \wedge, \vee$) whose deterministic generator must be doubly-exponential with doubly-exponential longest distance, that matches the upper bound for the full LTL. This is in sharp contrast to the fact that the longest distance of nondeterministic generators for LTL($\Box, \Diamond, \wedge, \vee$) formulas is only linear, and becomes exponential only by addition of next or until modalities.

2 Definitions

2.1 Linear Temporal Logic

We first recall the syntax and the semantics of linear temporal logic. We will define temporal logics by assuming that the atomic formulas are state predicates, that is, boolean combinations of atomic propositions. Given a set of atomic propositions, a *linear temporal logic* (LTL) formula is composed of state predicates, the boolean connectives *conjunction* ($\wedge$) and *disjunction* ($\vee$), the temporal operators *Next* ($\bigcirc$), *Eventually* ($\Diamond$), *Always* ($\Box$), and *Until* ($\mathcal{U}$). Formulas are built up in the usual way from these operators and connectives, according to the following grammar

$$\varphi := p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi \,\mathcal{U}\, \varphi.$$

An *ω-word* over a given alphabet Σ is a mapping from $\mathbb{N}$ into Σ, that is, an infinite sequence of symbols over Σ. LTL formulas are interpreted on an ω-word $w = w_0 w_1 w_2 \ldots$ over the alphabet $\Sigma = 2^P$ and the satisfaction relation $w \models \varphi$ is defined in the standard way. In the following, we will use the notation LTL($op_1, \ldots, op_k$) to denote the fragment of LTL which contains boolean combination of basic formulas which use only the boolean connectives and the temporal operators in the list $op_1, \ldots, op_k$.

2.2 Finite automata on ω-words

Automata on ω-words have been extensively studied in relation to temporal logic [8]. In this section, we will recall the definition of Büchi automata and the results relating them to LTL as *generators* of models.

A *nondeterministic transition graph* is a 4-tuple (Σ, S, S_0, Δ), where Σ is an alphabet, S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, and Δ is a subset of $S \times \Sigma \times S$. A transition graph is *deterministic* if $|S_0| = 1$ and Δ defines a total function δ from $S \times \Sigma$ into S. In the following, when we consider deterministic transition graphs, we will define directly this function δ instead of the transition relation Δ. The behavior of a transition graph on a word is captured by the concept of a *run*. Let $A = (\Sigma, S, S_0, \Delta)$ be a transition graph and w be an ω-word, a run of A on w is a mapping $r : \mathbb{N} \longrightarrow S$ such that $r(0) \in S_0$ and for all $i \in \mathbb{N}$, $(r(i), w(i), r(i+1)) \in \Delta$. Given a run r on a word w, we denote with $Inf(r)$ the set of states appearing infinitely often in r. A clear property of deterministic transition graphs is that they have exactly one run for each word.

Given a transition graph we define an automaton by specifying the acceptance conditions. A *nondeterministic (resp. deterministic) Büchi automaton* is a 5-tuple $A = (\Sigma, S, S_0, \Delta, F)$, where (Σ, S, S_0, Δ) is a nondeterministic (resp. deterministic) transition graph and $F \subseteq S$ is the set of the *accepting* states. An ω-word w is accepted by a Büchi automaton A iff there exists a run r of A on w such that $Inf(r) \cap F \neq \emptyset$. The language accepted by A, denoted by $L(A)$, is defined to be the set $\{w \mid w \text{ is accepted by } A\}$.

For our results, besides the size, another characterizing measure of an automaton A is the length of the longest simple directed path connecting two states in the transition graph. We will refer to this measure as the *longest distance* of A.

For every LTL formula φ, it is possible to construct an automaton on ω-words accepting all models of it. We will denote such an automaton as A_φ and we will refer to it as a *generator* of models for φ. A deterministic generator for an LTL formula of size $O(exp(exp(|\varphi|)))$ can be obtained in the following way: from the formula φ, by the tableau construction, it is possible to construct a nondeterministic Büchi generator of size $O(exp(|\varphi|))$ [12, 21]; this automaton can then be determinized so that we obtain a deterministic Rabin automaton of size $O(exp(exp(|\varphi|))$ [17]. Notice that in general, for a given formula φ, a deterministic Büchi generator may not exist but, when this exists, it has been proved that the translation from LTL formulas to deterministic Büchi automata is doubly-exponential [11], and thus, the above construction is asymptotically optimal.

2.3 Game graphs

In this section we will introduce the notation concerning two-player games. A two-player game is modeled by a *game graph* and a *winning condition*. A game graph is a tuple $G = (V, V_0, V_1, \Sigma, \gamma)$ where V

is a finite or countable set of vertices, V_0 and V_1 define a partition of V, Σ is a finite set of actions and $\gamma : V \times \Sigma \to V$ is a partial function. For $i = 0, 1$, the vertices in V_i are those from which only $Player_i$ can move and the allowed moves are given by the function γ. A winning condition is a predicate over ω-words of vertices, and depending on its type, we can have different kinds of games. In this paper we will consider only Büchi and LTL games. In a Büchi game, the winning condition is given by a set of vertices $F \subseteq V$ with the requirement that at least a state in F must repeat infinitely often. In an LTL game, the winning condition is instead an LTL formula.

A *play* of a game G is constructed as a sequence of vertices corresponding to the actions taken by the two players. Formally, a play starting at x_0 is a sequence $x_0 x_1 \dots x_h$ in V^* with the property that there exists a sequence of actions $a_1, \dots, a_h \in \Sigma$ such that $\gamma(x_{j-1}, a_j) = x_j$, for $j = 1, \dots, h$. Starting from a vertex u, a game G can be seen as the ω-tree $T_{(G,u)}$, called a *game tree*, which is obtained by unwinding G from u. Each node of this tree corresponds to a play starting at u: the root corresponds to u and, if a node v corresponds to a play $x_1 \dots x_h$, then each of its children corresponds to a possible continuation of the play $x_0 \dots x_h$, i.e. to a play $x_0 \dots x_h x_{h+1}$ such that $\gamma(x_h, a) = x_{h+1}$ for an action $a \in \Sigma$. A *strategy* for $Player_i$ gives an allowed move to continue each play ending at a vertex in V_i. More formally, a strategy for $Player_i$ is a total function $f : V^* V_i \to V$ mapping a node in the function domain into one of its successors in the game tree. A strategy then corresponds to a tree obtained from the game tree $T_{(G,u)}$ by pruning all the subtrees containing plays that are not constructed according to f. When a strategy depends only on the last vertex of a play, it is called a *memoriless strategy*.

Given a game G and a winning condition W, a strategy f is said to be a *winning strategy* if the requirement expressed by W holds on all the paths of the tree corresponding to f. In a two-player game, given a game G and a winning condition W, we consider the decision problem: "Is there a strategy for $Player_i$ satisfying the winning condition W?" We remark that while Büchi games admit memoriless winning strategies and can be solved in quadratic time, LTL games in general do not have a memoryless winning strategy and are decidable in time polynomial in $|G|$ and doubly-exponential in $|\varphi|$ [15].

3 Deterministic generators

We begin this section by introducing a proper subclass of deterministic Büchi automata whose transition function defines a partial order over the states. To emphasize this property, we call an automaton in this class a *partially-ordered deterministic Büchi automaton* (PODB). Then, we will show that, for formulas in some fragments of LTL, it is possible to construct a deterministic generator which is a PODB.

A PODB is a deterministic Büchi automaton whose transition graph is a directed acyclic graph except for the self-loops. Obviously, the longest distance of a PODB is the longest distance between the initial state and a sink state, where an initial and a sink state are respectively a minimal and a maximal state with respect to the partial order induced by the transition function of the PODB. PODBs are closed under boolean operations.

Proposition 3.1 *For $i = 1, 2$, let A_i be PODBs of size n_i and longest distance d_i. There exists a PODB $A_1 \cap A_2$ (resp. $A_1 \cup A_2$) accepting the language $L(A_1) \cap L(A_2)$ (respectively, $L(A_1) \cup L(A_2)$), and such that its size is $O(n_1 n_2)$ and its longest distance is not greater than $d_1 + d_2$. Moreover, for $i = 1, 2$, there exists a PODB $\overline{A_i}$ of size n_i and longest distance d_i accepting $\Sigma^* \setminus L(A_i)$.*

Note that to prove the above proposition, the construction for intersection does not require the introduction of a counter as in the case of general deterministic Büchi automata. Moreover, the above results on intersection and union are naturally extended to a tuple of automata $A_1, \dots, A_k$ and we will denote the corresponding automata with $A_1 \cap \dots \cap A_k$ and $A_1 \cup \dots \cup A_k$, respectively.

The following automaton construction will be used in the next sections to build the generator for $\Diamond(p \wedge \varphi)$ given the generator for φ. Let $A = (\Sigma, S, s_0, \delta, F)$ be a Büchi automaton and p be a predicate over Σ. Given a $s'_0 \notin S$, we define the (deterministic) Büchi automaton $A^{\Diamond(p,A)}$ as $(\Sigma, S \cup \{s'_0\}, s'_0, \delta', F)$ where:

- $\delta'(s, a) = \delta(s, a)$ for $s \in S$,
- $\delta'(s'_0, a) = \delta(s_0, a)$ for a satisfying p, and
- $\delta'(s'_0, a) = s'_0$, otherwise.

The construction is illustrated in Figure 1.

Proposition 3.2 *Let $A = (\Sigma, S, s_0, \delta, F)$ be a (deterministic) Büchi automaton of size n and longest distance d such that $\Sigma\, L(A) \subseteq L(A)$, and p be a predicate over Σ. The (deterministic) automaton $A^{\Diamond(p,A)}$ has size $O(n)$, longest distance $d + 1$ and accepts the language $\Sigma^* [p] L(A)$, where $[p] = \{a \in \Sigma \mid a \text{ satisfies } p\}$.*

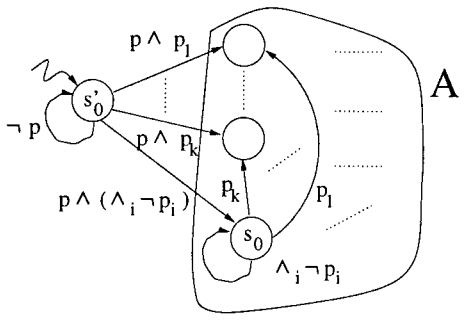


Figure 1: Graphical representation of the automaton $A^{\Diamond(p,A)}$.

Moreover, if A is a PODB then $A^{\Diamond(p,A)}$ is a PODB also.

3.1 Generators for LTL($\Diamond, \wedge$)

The fragment LTL($\Diamond, \wedge$) contains boolean combinations of formulas built from state predicates using eventualities and conjunctions. Thus, negations and disjunctions are allowed only at the top-level and at the atomic level. By definition, LTL($\Diamond, \wedge$) is equivalent to LTL($\Box, \vee$). A sample formula of this fragment is $\Box p \vee \Diamond(q \wedge \Diamond r)$. This fragment includes combinations of typical invariants and termination properties.

Let us consider the formula $\varphi = \Diamond p_1 \wedge \ldots \wedge \Diamond p_n$, where $p_i \in P$ for $i = 1, \ldots, n$. Obviously, φ is in LTL($\Diamond, \wedge$). This formula asserts that each one of $p_1, \ldots, p_n$ has to be true sometimes. Then, a deterministic generator A_φ for φ has to keep track only of the set of atomic propositions which have been already fulfilled. The size of A_φ is $O(2^n)$ and its longest distance is the cardinality of the maximal totally ordered set of states with respect to the subset relation, that is, n. We proceed to show that all the LTL($\Diamond, \wedge$) formulas have a deterministic generator which is a PODB of exponential size and linear longest distance, but first, we introduce a characterization of the formulas in the considered fragment. A formula φ in LTL($\Diamond, \wedge$) is a boolean combination of formulas defined inductively by the following rules:

- φ is a state predicate over P or,
- for $k \geq 0$, φ is $p \wedge \Diamond\varphi_1 \wedge \ldots \wedge \Diamond\varphi_k$ where p is a state predicate over P and $\varphi_1, \ldots, \varphi_k$ are formulas in LTL($\Diamond, \wedge$) that do not contain negations and disjunctions at the top-level.

Theorem 3.3 *There exists a deterministic Büchi automaton A accepting all the models of a formula φ in* LTL($\Diamond, \wedge$) *such that A is a PODB of $O(exp(|\varphi|))$ size and $O(|\varphi|)$ longest distance.*

Proof. We inductively define a deterministic Büchi automaton A accepting all the models of a given formula $\Diamond\varphi$ in LTL($\Diamond, \wedge$) such that A is a PODB of exponential size and linear longest distance in $|\varphi|$, and then by Proposition 3.1 this result is extended to a general formula in LTL($\Diamond, \wedge$). For a state predicate p, we define A_p and $A_{\Diamond p}$ as the minimal deterministic generator for p and $\Diamond p$, respectively. Clearly, A_p and $A_{\Diamond p}$ are PODBs and $A_{\Diamond p}$ is such that $\Sigma^* L(A_{\Diamond p}) \subseteq L(A_{\Diamond p})$. Now, let ψ be the formula $\Diamond(p \wedge \Diamond\psi_1 \wedge \ldots \wedge \Diamond\psi_k)$ and, for a formula $\gamma \in \{\psi_1, \ldots, \psi_k\}$, $A_{\Diamond\gamma}$ be a PODB accepting all the models of $\Diamond\gamma$. By inductive hypothesis we have that size of $A_{\Diamond\gamma}$ is $O(exp(|\Diamond\gamma|))$ and longest distance of $A_{\Diamond\gamma}$ is $O(|\Diamond\gamma|)$. Obviously, $\Sigma^* L(A_{\Diamond\gamma}) \subseteq L(A_{\Diamond\gamma})$ also holds. Then, by Proposition 3.1, $A' = A_{\Diamond\psi_1} \cap \ldots \cap A_{\Diamond\psi_k}$ is a PODB of $O(exp(|\Diamond\psi_1| + \ldots + |\Diamond\psi_k|))$ size, $O(|\Diamond\psi_1| + \ldots + |\Diamond\psi_k|)$ longest distance, and such that $\Sigma^* L(A') \subseteq L(A')$. Thus, from Proposition 3.2, we have that $A_\psi = A^{\Diamond(p,A')}$ is the generator for ψ. ∎

The previous result is optimal in the sense that we may not have a smaller generator for some formula in LTL($\Diamond, \wedge$), as shown in the following theorem.

Theorem 3.4 *There exists a formula φ in* LTL($\Diamond, \wedge$) *such that all generators of φ have $\Omega(exp(|\varphi|))$ size and $\Omega(|\varphi|)$ longest distance.*

Proof. Consider the formula $\varphi = \Diamond p_1 \wedge \ldots \wedge \Diamond p_n$, where $p_i \in P$ for $i = 1, \ldots, n$ and $n \geq 2$. Clearly, $|\varphi| = O(n)$. The first assertion can be easily proved by contradiction showing that the initial state of a φ generator must have at least $2^n - 1$ successors. The second assertion can be proved by contradiction by showing that if a generator A_φ for φ has longest distance less than n, from the φ model $w = \{p_1\}\{p_2\}\ldots\{p_n\}^\omega$, we can derive another word which is not a model of φ but is accepted by A_φ. ∎

3.2 Generators for LTL($\Diamond, \bigcirc, \wedge$)

In this section we use the notation $\bigcirc^n$ as a shorthand for n nested next modalities. We therefore consider size of $\bigcirc^n \varphi$ to be $|\varphi| + n$. Let us consider the formula $\varphi = \Diamond(p \wedge \bigcirc^n q)$, where $p, q \in P$. This formula asserts that p has to be fulfilled at a position i and q at a position $i + n$ for some $i \in \mathbb{N}$. A deterministic generator for φ has to keep track of the truth values of p in the previous n positions. This can be done by running n copies of the deterministic generators for $(p \wedge \bigcirc^n q)$. Such a generator requires exponentially many states and has exponential longest distance. We prove that this upper bound holds for all LTL($\Diamond, \bigcirc, \wedge$) formulas:

Theorem 3.5 *There exists a deterministic Büchi automaton A accepting all the models of a formula φ in* LTL($\Diamond$,$\bigcirc$,$\wedge$) *such that A has both size and longest distance at most exponential in $|\varphi|$.*

Proof. The construction is done inductively on the structure of formulas in LTL($\Diamond$,$\bigcirc$,$\wedge$). We observe that given a formula ψ, the next operators in ψ can be pushed inside so that we can obtain an equivalent formula ψ' having only state predicates in the scope of a finite sequence of next operators, and such that $\psi' = O(|\psi|^2)$. As a consequence most of the cases are handled as for the construction of a deterministic generator for LTL($\Diamond$,$\wedge$) formulas. The interesting case is to construct a deterministic generator for $\varphi = \Diamond(p \wedge \bigcirc^k q \wedge \varphi')$ given a deterministic generator $A_{\varphi'}$ for φ' of both size and longest distance exponential in $|\varphi|$, and such that $\Sigma^* L(A_{\varphi'}) \subseteq L(A_{\varphi'})$. A deterministic generator A_φ for φ can be obtained by running in parallel k copies of $A_{\varphi'}$ and checking for the fulfillment of $(p \wedge \bigcirc^k q)$. At every position i of the input word a copy of $A_{\varphi'}$ is started and if $i > k$ and $(p \wedge \bigcirc^k q)$ is not true at position $(i-k)$ then the copy started at position $(i-k)$ is dismissed. As soon as $(p \wedge \bigcirc^k q)$ becomes true, A_φ dismisses all copies of $A_{\varphi'}$ but the one started at the position where $(p \wedge \bigcirc^k q)$ is true, and continues as $A_{\varphi'}$. The size of A_φ is thus $O(exp(k\,|P|)|A_{\varphi'}|)$ and hence exponential in $|\varphi|$. Its longest distance is $O(exp(k) + d')$, where d' is the longest distance of $A_{\varphi'}$, and thus is exponential in $|\varphi|$. ∎

The previous result is optimal in the sense that we may not have a smaller generator for some formula in LTL($\Diamond$,$\bigcirc$,$\wedge$), as shown in the following theorem.

Theorem 3.6 *There exists a formula φ in* LTL($\Diamond$,$\bigcirc$,$\wedge$) *such that all generators of φ have $\Omega(exp(|\varphi|))$ size and $\Omega(exp(|\varphi|))$ longest distance.*

Proof. Consider the formula $\varphi = \Box(p \rightarrow \bigcirc^n q)$, where $p, q \in P$ and $n \geq 2$. Clearly, $|\varphi| = O(n)$. Since LTL($\Diamond$,$\wedge$) is a fragment of LTL($\Diamond$,$\bigcirc$,$\wedge$), we only need to prove that all generators for φ have a simple path of length at least 2^n. Assume that $A_\varphi = (2^P, S, s_0, \Delta, F)$ is a generator for φ. Consider words $w = a_1 \ldots a_n$ and $w' = a'_1 \ldots a'_n$ such that $w, w' \in (2^P)^*$, and $p \notin a_i$ and $p \in a'_i$ for some i. Let $y \in (2^P)^\omega$ be such that $y = b_1 \ldots b_h \ldots$, $q \notin b_i$, and xwy is a model of φ for some $x \in (2^P)^*$. We have that $xw'y$ is not a model of φ. Thus a generator A_φ cannot enter the same state after reading xw and xw', since it must accept xwy and reject $xw'y$. Clearly we can prove this for any pair of words w, w' of length n that differs with respect to the truth of p at least in a position. Since we can determine 2^n words $w_1, \ldots, w_{2^n}$ which are pairwise different with respect to truth values of p, there are 2^n pairwise disjoint sets of states each of them contains the states which are reached on all runs of A_φ by reading a prefix of a model for φ ending in w_i. To conclude this proof we just need to prove that there exists a word that forces A_φ to visit a state from each of these sets without reentering any of them before reading at least one state from each set. But this is equivalent to prove that there is an exponentially long word w in $\{0,1\}^*$ such that any two subwords of w of length n differ at least in a position, and thus we are done. ∎

3.3 Generators for LTL($\Diamond$,$\wedge$,$\vee$) and LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$)

The fragment LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$) contains boolean combinations of formulas built from state predicates using eventualities, next, disjunctions, and conjunctions. This fragment includes combinations of safety and guarantee properties, and belongs to the class of syntactic obligation properties [13].

Let us consider the formula $\varphi = \Diamond \bigwedge_{i=1}^{n} (p_i \vee \Diamond q_i)$, where $p_i, q_i \in P$, for $i = 1, \ldots, n$ and $n \geq 2$. Obviously φ is an LTL($\Diamond$,$\wedge$,$\vee$) formula. This formula asserts that at a same position in the model all the clauses $(p_i \vee \Diamond q_i)$ have to be satisfied. Since the fulfillment of a clause at a position implies either $p_i \vee q_i$ at that position or q_i at a later position, a nondeterministic generator for φ is the one that nondeterministically guesses the first position at which all the clauses are satisfied and, then, check for their fulfillment. Such a generator has an exponential size and a linear longest distance. We can determinize this strategy to obtain a deterministic generator for φ with $O(2^{2^n})$ states and $O(2^n)$ longest distance. It is possible to prove that this result indeed holds for all LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$) formulas, as stated by the following theorem.

Theorem 3.7 *There exists a deterministic Büchi automaton A accepting all the models of a formula φ in* LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$) *such that A has size doubly-exponential in $|\varphi|$ and longest distance exponential in $|\varphi|$.*

Proof. To construct a deterministic generator for LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$) formulas we first transform them into a "layered" conjunctive normal form where we have either LTL($\Diamond$,$\bigcirc$,$\wedge$) formulas or formulas of type $\psi = \Diamond \bigvee_i (p_i \wedge \bigcirc^k q_i \wedge \psi_i)$. This translation may cause an exponential blow-up in the size of the formula. The results obtained for LTL($\Diamond$,$\bigcirc$,$\wedge$) then give the upper bound on the size of the deterministic genera-

tor for LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$) formulas. An accurate analysis of the longest distance in the construction given for LTL($\Diamond$,$\bigcirc$,$\wedge$) gives an $O(exp(k\,|P|) + |\psi|)$ upper bound, where k is the largest number of nested next modalities in the starting formula. Since the transformation into the layered CNF does not increase this parameter, given an LTL($\Diamond$,$\bigcirc$,$\wedge$,$\vee$) formula we get that the longest distance of the deterministic generator obtained by the given construction is exponential in $|\varphi|$. ∎

The following theorem shows that the above result is optimal also in the case of LTL($\Diamond$,$\wedge$,$\vee$) formulas.

Theorem 3.8 *There exists a formula φ in* LTL($\Diamond$,$\wedge$,$\vee$) *such that all the deterministic generators of φ have $\Omega(exp(exp(|\varphi|)))$ size and $\Omega(exp(|\varphi|))$ longest distance.*

Proof. Consider the formula $\varphi = \Diamond \bigwedge_{i=1}^{n}(p_i \vee \Diamond q_i)$, where $p_i, q_i \in P$ for $i = 1, \ldots, n$ and $n \geq 2$. Obviously, $|\varphi| = O(n)$. Denote with P_p the set $\{p_1, \ldots, p_n\}$ and P_q the set $\{q_1, \ldots, q_n\}$. We prove that a minimal deterministic generator for φ has $2^{2^{\Omega(n)}}$ states. With a similar argument it is also possible to show that all the deterministic generators for φ have a simple path of length $2^{\Omega(n)}$. Assume that $A_\varphi = (2^P, S, s_0, \delta, F)$ is a deterministic generator for φ. Given a subset b of P_p, define $q(b)$ as the set $\{q_i \,|\, p_i \notin b\}$. Define Σ_k as the set of P_p subsets of cardinality k, that is, $\Sigma_k = \{a \subseteq P_p \,|\, |a| = k\}$. The cardinality of Σ_k is $\binom{n}{k}$. If we choose $k = \lceil \frac{n}{2} \rceil$, then $|\Sigma_k| = 2^{\Omega(n)}$. Observe that for $w, w' \in \Sigma_k^*$ such that $w = \sigma_0\sigma_1 \ldots \sigma_m$, $w' = \sigma'_0\sigma'_1 \ldots \sigma'_{m'}$ and $\cup_{i=1}^{m}\{\sigma_i\} \neq \cup_{i=1}^{m'}\{\sigma'_i\}$, it must hold that $\delta(s_0, w) \neq \delta(s_0, w')$. In fact, we can suppose without loss of generality that there is a $\sigma \in \cup_{i=1}^{m}\{\sigma_i\}$ such that $\sigma \notin \cup_{i=1}^{m'}\{\sigma'_i\}$. Thus, for any $w'' \in (2^P)^\omega$, the word $wq(\sigma)w''$ is a model of φ and $w'q(\sigma)\emptyset \ldots \emptyset \ldots$ is not. Since A_φ accepts all and only the models of φ, and there is an accepting run for any word $wq(\sigma)w''$, if $\delta(s_0, w) = \delta(s_0, w')$ then A_φ accepts also $w'q(\sigma)\emptyset \ldots \emptyset \ldots$, and this contradicts the hypothesis A_φ being a generator of models for φ. Since the number of subsets of Σ_k is $2^{|\Sigma_k|}$, A_φ must have at least $2^{|\Sigma_k|}$ states. Thus, for $k = \lceil \frac{n}{2} \rceil$, this means $2^{2^{\Omega(n)}}$ states. ∎

3.4 Generators for LTL($\Box$,$\Diamond$,$\wedge$,$\vee$)

In section 2.2 we recalled the results concerning the construction of a deterministic generator for a given formula in LTL. In this section we prove that a matching lower bound to that construction even in absence of next and until modalities.

Theorem 3.9 *There exists a formula φ in* LTL($\Box$,$\Diamond$,$\wedge$,$\vee$) *such that all the deterministic generators of φ have an $\Omega(exp(exp(|\varphi|))$ longest distance.*

Proof. Consider the formula

$$\Box(\Diamond \bigwedge_{i=1}^{n}(a_i \vee \Diamond b_i) \rightarrow \Diamond \bigwedge_{i=1}^{n}(c_i \vee \Diamond d_i)),$$

where $a_i, b_i, c_i, d_i \in P$ for $i = 1, \ldots, n$ and $n \geq 2$. Assume that $A_\varphi = (2^P, S, s_0, \delta, F)$ is a deterministic generator for φ. Denote by P_x the set $\{x_1, \ldots, x_n\}$. Moreover, denote by p_j a subset of P_a and by q_j a subset of P_c. By arguments similar to those used in the proof of Theorem 3.8, it is possible to prove that: 1) a deterministic generator for φ has to keep track of the p_j's that have been fulfilled and for each p_j the list of q_h's which have been fulfilled starting at the position where p_j was true the last time; 2) we may need to store exponentially many p_j's and exponentially many q_j's, to check the fulfillment of $\Diamond \bigwedge_{i=1}^{n}(a_i \vee \Diamond b_i)$ and $\Diamond \bigwedge_{i=1}^{n}(c_i \vee \Diamond d_i)$, respectively. Thus for $k = \Omega(2^n)$, let $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$ such sets. We observe that only one among all p_j's (respectively, q_j's) can be true at each position. Every time a p_j is true at a position i, A resets the list for p_j with only the q_h which is true at position i. Every time a q_j is true, A adds q_j to all lists. To conclude the proof it is sufficient to show that there exists a word w in $(P_p \cup P_q \cup \{p_j \cup q_h \,|\, p_j \in P_p, q_h \in Q_p\})^*$ of length 2^k such that the A run on w is such that $r(i) \neq r(j)$ for any $i \neq j$. To see this, we map each state s of A into a binary k-tuple $(x_1, \ldots, x_k)$ such that $x_i = 1$ if and only if q_i is in the list for p_i. Clearly, if two states s and s' are mapped into two different tuples then $s \neq s'$. Moreover, by the above observations, if neither q_i or p_i is true at the current position the i-th bit of the tuple associated to the next A state is the i-th bit of the current state, while if q_i true then the i-th bit becomes 1, otherwise if p_i is true the i-th bit becomes 0. Since at most a p_i and a q_j are true at each position, the tuples of two consecutive states in a run may differ for at most 2 bits. Since it is possible to list all the 2^k binary tuples in such a way two consecutive tuples differs in exactly 1 or 2 bits, we have proved that any deterministic generator for φ has $\Omega(2^k) = \Omega(2^{2^n})$ longest distance. ∎

4 Büchi games

In this section we present a new decision algorithm for Büchi games, which mainly performs a depth-first

traversal of a portion of the game tree and is space-efficient when the longest distance is $O(\frac{n}{\log n})$. Standard techniques to solve Büchi games involve fix-point computation [19], and requires space $O(n)$ no matter what the longest distance is. An interesting aspect of our algorithm is that it can be applied to all the games in which the winning condition can be translated into a deterministic Büchi automaton, as for the formulas in the fragments of LTL we have studied in sections 3.1, 3.2 and 3.3. Then we combine this algorithm with the results on LTL generators from the previous section and study the complexity of the obtained solutions.

In this section we search for winning strategies of $Player_0$, while $Player_1$ will be our adversary. Consider a game graph G and a subset F of G vertices. We denote by Π the set of plays whose last state is the first state which repeats, that is, plays of the form $x_0 \dots x_h$ such that $x_h = x_i$ for some $0 \leq i < h$, and for all $0 \leq i, j < h$, $x_i \neq x_j$. We have that any long-enough play in G has a prefix which is in Π, and each of the plays from Π is constituted by an acyclic prefix followed by a loop. Moreover, we denote by Π_F the set of plays in Π containing a state from F in their loop, and by Π_f the set of plays from Π which can be constructed using the strategy f. We define a game $(G, F)_{fin}$ as the game where $Player_0$ wins from a state u if there is a strategy f from u such that $\Pi_f \subseteq \Pi_F$. Since Büchi games are memoryless, we have:

Lemma 4.1 *There exists a winning strategy for $Player_0$ from a vertex u in a Büchi game (G, F) if and only if there exists a winning strategy for $Player_0$ from u in $(G, F)_{fin}$.*

Directly from the definition of a winning strategy in a game $(G, F)_{fin}$, we have the following lemma.

Lemma 4.2 *Any winning strategy f for $Player_0$ in a game $(G, F)_{fin}$ is such that the lenght of a play in Π_f is $O(d)$, where d is the longest distance of G.*

By the above lemmas, there is a decision algorithm for Büchi games which explores a tree whose height is the longest distance of the game graph.

Theorem 4.3 *Given a game graph G with m vertices and longest distance d, the Büchi game (G, F) is decidable in space $O(d \log m)$.*

Given a game (G, W), if the winning condition W can be translated to a deterministic Büchi automaton, it is possible to use the algorithm by Lemmas 4.1 and 4.2 to decide it. In particular, let A be a deterministic Büchi automaton equivalent to winning condition W, in the sense that the language accepted by A is the language of the ω-words satisfying W. Define $G \times A$ as the game graph whose vertices $V \times Q$, where Q is the set of A states, are partitioned according to the V partition, and from a vertex (v, q) it is possible to reach a vertex (v', q') by taking an action a if and only if A enters q' from q by reading the subset of atomic propositions true at v and in G it is possible to move from v to v' taking the action a. Let F and s_0 be the set of final states and the initial state of A, respectively, then there is a winning strategy in the Büchi game $(G \times A, V \times F)$ starting at a vertex (u, s_0) if and only if there is a winning strategy in (G, W) starting at u.

As a consequence of the results from section 3 and the above argument, Theorem 4.3 applies to games with winning condition expressed by formulas in the LTL fragments we have considered so far. In fact, the following theorems hold.

Theorem 4.4 LTL$(\Diamond, \wedge)$ *games are* PSPACE*-complete.*

Proof. Membership in PSPACE is a consequence of Theorems 3.3 and 4.3. To prove PSPACE-hardness, we can reduce the satisfiability of quantified boolean formulas in conjunctive normal form to deciding the existence of a winning strategy in an LTL$(\Diamond, \wedge)$ game. This also shows that LTL$(\Box, \vee)$ games are PSPACE-hard. Let $\varphi = A_1 x_1 \dots A_n x_n . \bigwedge_{i=1}^{m} c_i$ be a quantified boolean formula over the variables $x_1, \dots, x_n$. Consider the LTL$(\Diamond, \wedge)$ formula $\varphi' = \bigwedge_{i=1}^{m} \Diamond c_i$ over the atomic propositions $\{c_1, \dots, c_m\}$. The game graph G is defined in such a way that each literal corresponds only to a vertex, a path of the game tree corresponds to the assignment given by assuming true the literals corresponding to its vertices, each vertex is labeled with the conjuncts which contain the corresponding literal, and a strategy corresponds to a selection of paths fulfilling the requirements of quantifiers $A_1, \dots, A_n$. We have that φ is satisfiable if and only if there is a winning strategy in the game (G, φ'). ∎

Theorem 4.5 LTL$(\Diamond, \bigcirc, \wedge)$ *games are* EXPTIME*-complete.*

Proof. By Theorem 3.5, LTL$(\Diamond, \bigcirc, \wedge)$ has exponentially-sized deterministic generators, and hence, membership in EXPTIME follows. For the lower bound, we reduce the halting problem for alternating linear bounded automata. We briefly sketch the construction. Consider a Turing machine M that uses n tape positions over a tape alphabet Γ, and let Q be the set of control states that are partitioned into Q_0 and Q_1 corresponding to the two players. The transitions of the machine are of the form $\langle q, \sigma, q', \sigma', L/R \rangle$ meaning that if control state is q and current symbol is σ, then the machine can

overwrite the current cell with σ', update control state to q', and move left (L) or right (R). If multiple transitions are applicable, then depending on whether the current control state belongs to Q_0 or Q_1, one of the two players gets to choose the transition. The problem of deciding whether $Player_0$ has a strategy to reach a specified control state, say q_h, is EXPTIME-complete. Given such a machine M, we build a game graph G_M as follows. For every tape symbol σ and position i, G_M has a vertex $v_{\sigma,i}$ belonging to V_1. For every control state q, tape symbol σ and position i, G_M has a vertex $v_{q,\sigma,i}$ belonging to V_0 if q is in Q_0 and to V_1 otherwise. For every control state q, and symbol σ, G_M has a vertex $v_{q,\sigma,L}$ and a vertex $v_{q,\sigma,R}$, both belonging to V_1. For $i < n$, there is an edge from $v_{\sigma,i}$ to every $v_{\sigma',i+1}$. There is an edge from $v_{\sigma,n}$ to every $v_{q,\sigma',i}$. For every transition $\langle q, \sigma, q', \sigma', L/R\rangle$ of M, there is an edge from every $v_{q,\sigma,i}$ to $v_{q',\sigma',L/R}$. Finally, every $v_{q,\sigma,L/R}$ has an edge to every $v_{\sigma',1}$. The intuition is that $Player_1$ chooses a sequence of vertices $v_{\sigma_1,1}, \ldots v_{\sigma_n,n}$, denoting the tape content, followed by a vertex $v_{q,\sigma,i}$, meaning that current control is in state q with head reading symbol σ in position i. The next vertex of the form $v_{q',\sigma',L/R}$ indicates the choice of the transition (and hence, new control state and new symbol in position i, and movement of the head), and is determined by one of the players depending on whether q belongs to Q_0 or Q_1. $Player_0$ wins if either the control state q_h is encountered or $Player_1$ does not make the choices for encoding the configuration according to the intended interpretation. Assume that there are enough propositions to identify each vertex uniquely by a state predicate. Then, the winning condition for $Player_0$ is a top-level disjunction of several formulas that use only eventualities and conjunctions. For instance, a mistake in the encoding of the content of i-th tape position is described by the formula

$$\vee \Diamond(v_{\sigma,i} \wedge \bigcirc^{n-i+1} v_{q,\sigma',i' \neq i} \wedge \bigcirc^{n+2} v_{\sigma'' \neq \sigma,i})$$

$$\vee \Diamond(v_{\sigma,i} \wedge \overset{n-i+1}{\bigcirc} v_{q,\sigma,i} \wedge \overset{n-i+2}{\bigcirc} v_{q',\sigma',L/R} \wedge \overset{n+2}{\bigcirc} v_{\sigma'' \neq \sigma',i})$$

∎

Theorem 4.6 LTL($\Diamond, \bigcirc, \wedge, \vee$) *games are* EXPSPACE.

Proof. Directly from Theorems 3.7 and 4.3. ∎

5 Conclusions

For the problem of solving infinite games with the winning condition specified by an LTL formula, we have studied the impact of different connectives on the complexity. In the same way as model checking (or satisfiability) is related to translation from LTL to nondeterministic ω-automata, solving games is related to translation from LTL to deterministic ω-automata. We have established that the longest distance, besides the size, of the automaton produced by the translation is an important parameter. The results are summarized in the table of Figure 2 for various fragments [2]. As the table indicates the sources of complexity for games are different from the ones for model checking. The matching lower bounds for the games in the LTL fragments LTL($\Diamond, \wedge, \vee$), LTL($\Diamond, \bigcirc, \wedge, \vee$), and LTL($\Box, \Diamond, \wedge, \vee$) are open problems, while the results on the corresponding deterministic generators are tight with respect to both the size and the longest distance. We observe that LTL($\Box, \Diamond, \wedge, \vee$) and thus LTL, formulas may not have deterministic Büchi generators, but it is known that they have doubly-exponential deterministic Streett generators.

Besides the classification of complexity of games for various fragments, the constructions of this paper can be used to solve synthesis problems for certain kinds of formulas more efficiently. In particular, the fragments LTL($\Diamond, \wedge$) and LTL($\Diamond, \wedge, \vee$) contains many commonly occuring specifications that are boolean combinations of safety and guarantee properties, and for these, we have provided a direct construction of deterministic generators in a modular manner.

References

[1] R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *CONCUR'99: Concurrency Theory, Tenth Int. Conference*, LNCS 1664, pages 82–97, 1999.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. of the 38th IEEE Symposium on Foundations of Computer Science*, pages 100 – 109, 1997.

[3] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the Tenth Int. Conference on Computer Aided Verification*, LNCS 1427, pages 521 – 525. Springer-Verlag, 1998.

[4] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reac-

[2] Some of the entries in the table concerning nondeterministic generators and model checking are not explicitly stated in the literature, and will be explained in detail in the full paper.

	Nondet. Generators		Det. Generators		Model Checking	Games
	Size	Long. Distance	Size	Long. Distance		
LTL($\Diamond, \wedge$)	Θ(EXP)	Θ(LINEAR)	Θ(EXP)	Θ(LINEAR)	NP-complete	PSPACE-complete
LTL($\Diamond, \bigcirc, \wedge$)	Θ(EXP)	Θ(EXP)	Θ(EXP)	Θ(EXP)	PSPACE-complete	EXPTIME-complete
LTL($\Diamond, \wedge, \vee$)	Θ(EXP)	Θ(LINEAR)	Θ(2EXP)	Θ(EXP)	NP-complete	EXPSPACE
LTL($\Diamond, \bigcirc, \wedge, \vee$)	Θ(EXP)	Θ(EXP)	Θ(2EXP)	Θ(EXP)	PSPACE-complete	EXPSPACE
LTL($\Box, \Diamond, \wedge, \vee$)	Θ(EXP)	Θ(LINEAR)	Θ(2EXP)	Θ(2EXP)	NP-complete	2EXPTIME
LTL	Θ(EXP)	Θ(EXP)	Θ(2EXP)	Θ(2EXP)	PSPACE-complete	2EXPTIME-complete

Figure 2: Complexity results in LTL.

tive systems. In *Proc. of the 16th Int. Colloquium on Automata, Languages and Programming, ICALP'89*, LNCS 372, pages 1–17, 1989.

[5] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* ACM Distinguished Dissertation Series. MIT Press, 1989.

[6] S. Demri and Ph. Schnoebelen. The complexity of propositional linear temporal logics in simple cases. In *Proc. of the 15th Annual Symposium on Theoretical Aspects of Computer Science, STACS'98*, LNCS 1373, pages 61 - 72. Springer-Verlag, 1998.

[7] E.A. Emerson and C.S. Jutla. The complexity of tree automata and logics of programs. In *Proc. of the 29th IEEE-CS Symposium on Foundations of Computer Science*, pages 328 - 337, 1988.

[8] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, pages 995 - 1072. Elsevier Science Publishers, 1990.

[9] O. Kupferman and M.Y. Vardi. Module checking. In *Computer Aided Verification, Proc. Eighth Int. Workshop*, LNCS 1102, pages 75 - 86. Springer-Verlag, 1996.

[10] O. Kupferman and M.Y. Vardi. Module checking revisited. In *Proc. of the Ninth Int. Conference on Computer Aided Verification, CAV'97*, LNCS 1254, pages 36 -47, 1997.

[11] O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. of the 13th IEEE Symposium on Logic in Computer Science*, pages 81 - 92, 1998.

[12] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 97 - 107, 1985.

[13] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer-verlag, 1991.

[14] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46 - 77, 1977.

[15] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages*, pages 179 - 190, 1989.

[16] M.O. Rabin. Automata on infinite objects and Church's problem. *Trans. Amer. Math. Soc.*, 1972.

[17] S. Safra. On the complexity of ω-automata. In *Proc. of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319 - 327, 1988.

[18] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *The Journal of the ACM*, 32:733 - 749, 1985.

[19] W. Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science, STACS'95*, LNCS 900, pages 1 - 13. Springer-Verlag, 1995.

[20] M.Y. Vardi. Verification of concurrent programs: the automata-theoretic framework. In *Proc. of the Second IEEE Symposium on Logic in Computer Science*, pages 167 - 176, 1987.

[21] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1 - 37, 1994.

Session 8

Normalization by evaluation for typed lambda calculus with coproducts

T. Altenkirch[*] P. Dybjer[†] M. Hofmann[‡] P. Scott[§]

Abstract

We solve the decision problem for simply typed lambda calculus with strong binary sums, equivalently the word problem for free cartesian closed categories with binary coproducts. Our method is based on the semantical technique known as "normalization by evaluation" and involves inverting the interpretation of the syntax into a suitable sheaf model and from this extracting appropriate unique normal forms. There is no rewriting theory involved, and the proof is completely constructive, allowing program extraction from the proof.

1 Introduction

In this paper we solve the decision problem for simply typed lambda calculus with categorical coproduct (strong disjoint sum) types. While this calculus is both natural and simple, the decision problem is a long-standing thorny issue in the subject. Our solution is based on normalization by evaluation (NBE) (also called "reduction-free normalisation") introduced by Martin-Löf [ML75] for weak typed lambda calculus, and by Berger and Schwichtenberg [BS91] for typed lambda calculus with $\beta\eta$-conversion. The technique has been further refined by the authors and coworkers using category-theoretic methods [CD97, AHS95, CDS97]. It has also been extended to other systems, such as System F [AHS96]. As shown by Berger, Eberl, Schwichtenberg, and Danvy [BES98, Da96], NBE techniques yield fast normalization algorithms, with applications in interactive proof systems [BBSSZ98] and type-directed partial evaluation [Da96, Da98, Fil01].

Here we show how to considerably extend the NBE techniques to take into account type systems with strong sums. The NBE method involves constructing a model $\mathcal{M}$ and effectively "inverting" the evaluation of lambda terms in $\mathcal{M}$ and thereby extracting certain unique syntactic normal forms, from which a decision procedure easily follows (we outline the proof below). The proof uses no rewriting theory.

Typed lambda calculi with (strong) sum types arise very naturally:

- In programming language theory, coproducts model variant and enumerative types. The added categorical equation for coproducts corresponds to a kind of uniqueness for *pattern matching* or `Case` construction [AC98, Mit96, GLT89].
- In proof theory, under the Curry-Howard Isomorphism, terms correspond to natural deduction proofs in intuitionistic propositional $\{\wedge, \vee, \Rightarrow, \top\}$ logic. One then considers terms (proofs) modulo certain equations, which guarantee, for example, that the formula $A \vee B$ acts as a coproduct type (with copairing), as well as including the theory of commutative conversions (cf [GLT89], pp 80-81). In category theoretic terminology, such lambda theories correspond exactly to *almost bicartesian closed categories*, that is, cartesian closed categories with nonempty finite coproducts (generated by a set of atomic types) [LS86].
- As proved by Dougherty and Subrahmanyam [DS95], a Friedman completeness theorem in **Set** holds for cartesian closed categories with binary coproducts. Therefore, the equality we decide is the natural extensional equality on proofs in intuitionistic propositional logic and on terms of the typed lambda calculus with sums.

Much of traditional lambda calculus theory carries through unscathed when we add products (and even weak categorical data types) to the simply typed case. Unfortunately, the addition of coproducts is considerably more subtle. The difficulties with adding coproducts are detailed in [Do93, DS95]: for example, the analog of Statman's 1-Section theorem fails in the presence of coproducts, confluence (of various standard rewriting

[*] School of Computer Science and IT, University of Nottingham, Nottingham, UK, *e-mail:* txa@cs.nott.ac.uk

[†] Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden, *e-mail:* peterd@cs.chalmers.se

[‡] Division of Informatics (LFCS), University of Edinburgh, Edinburgh EH9 3JZ, UK, *e-mail:* mxh@dcs.ed.ac.uk

[§] Department of Mathematics, University of Ottawa, Ottawa, Ontario, K1N 6N5, Canada. *e-mail:* phil@mathstat.uottawa.ca.

The author's research is supported by a grant from NSERC.

0-7695-1281-X/01 $10.00 © 2001 IEEE

presentations) fails, and the proof of Friedman's completeness theorem for the case of coproducts uses difficult and involved syntactical arguments [DS95].

A decision procedure for cartesian closed categories with binary coproducts has been presented in Ghani's thesis [Gh95a] (see [Gh95b] for a summary) although the proof involves intricate rewriting techniques whose details are daunting. Our method described here is quite different and we believe conceptually simpler.

An algorithm for type-directed partial evaluation for a call-by-value typed lambda calculus with sums has been given by Danvy [Da96, Da98] and Filinski [Fil01]. This algorithm uses continuations and is therefore also quite different from ours. In particular, it does not decide equality in cartesian closed categories with binary coproducts.

Like Ghani and Dougherty and Subrahmamyam, we only consider the case of finite *non-empty* coproducts, that is, an initial object (empty type) is not part of the structure. We conjecture that the present approach can be extended to full bicartesian closed categories including initial objects. However, this complicates the structure of our normal forms, and we have not yet completely checked that all properties hold for the extended language.

Outline of Proof

Let $\mathcal{E}$ be a lambda theory. Our aim is to decide if

$$\Gamma \vdash_{\mathcal{E}} e_1 = e_2 : A,$$

that is, if two possibly open terms e_1 and e_2 of type A are equal wrt $\mathcal{E}$, where Γ is a type environment . We associate with each term e a *normal form* $\mathsf{nf}(e)$. In this paper, these normal forms are not themselves terms, but there is a function d mapping normal forms to terms in such a way that the following two properties hold (cf. [CD97, CDS97]):

NF1 $\Gamma \vdash_{\mathcal{E}} \mathsf{d}(\mathsf{nf}(e)) = e$

NF2 $\Gamma \vdash_{\mathcal{E}} e_1 = e_2$ implies $\mathsf{nf}(e_1) = \mathsf{nf}(e_2)$.

This implies that $\Gamma \vdash_{\mathcal{E}} e_1 = e_2$ if and only if $\mathsf{nf}(e_1) = \mathsf{nf}(e_2)$, so that comparing normal forms will yield a decision procedure for $\mathcal{E}$.

When $\mathcal{E}$ = the typed lambda calculus with $\beta\eta$-conversion, the authors and coworkers showed in [AHS95, CDS97] how to obtain a function nf by inverting the presheaf interpretation of $\mathcal{E}$. One defines two natural transformations $\mathsf{q}^A : [\![A]\!] \to \mathrm{NF}(A)$ and $\mathsf{u}^A : \mathrm{NE}(A) \to [\![A]\!]$, where $\mathrm{NF}(A)$ is the presheaf of normal forms and $\mathrm{NE}(A)$ is the presheaf of neutral terms of type A from $\mathcal{E}$. Given a typing judgement $\Gamma \vdash_{\mathcal{E}} e : A$, where $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we define

$$\mathsf{nf}(e) = \mathsf{q}([\![e]\!](\mathsf{u}(1_\Gamma)))$$

where 1_Γ is the sequence $(x_1, \ldots, x_n)$ and we omit type superscripts. Since $[\![-]\!]$ is an interpretation, we have immediately that $\Gamma \vdash e_1 = e_2$ implies $[\![e_1]\!] = [\![e_2]\!]$, and hence NF2 follows and NF1 is proved by induction on e, using for example logical relations.

How do we obtain a function nf when we add strong sums to $\mathcal{E}$? The problem is that although the category of presheaves has coproducts, a difficulty arises when we try to invert the interpretation of coproducts. The maps q and u are defined by induction on types, so in particular we need to define $\mathsf{u}^{A_0+A_1}$ in terms of u^{A_0} and u^{A_1}. But coproducts in presheaves are calculated pointwise; so, for example, how do we define $\mathsf{u}^{A_0+A_1}(s) \in [\![A_0]\!]_\Gamma + [\![A_1]\!]_\Gamma$ for a neutral term $\Gamma \vdash s : A_0 + A_1$? Since variables are neutral terms, we must in particular define $\mathsf{u}^{A_0+A_1}(x)$, but there is no sensible way to decide whether this should be in the first or the second disjunct.

As we shall show, the solution of this problem is to introduce an appropriate Grothendieck topology and consider the sheaves for that topology. This will give us a way to "amalgamate" the contributions of u^{A_0} and u^{A_1} in the definition of $\mathsf{u}^{A_0+A_1}$.

Plan of the paper

In Section 2 we formally define the typed lambda calculus with strong sums and show how it yields a free cartesian closed category with binary coproducts. In Section 3 we introduce our normal forms, and the auxiliary notions of pure normal forms and neutral terms. The main idea is to introduce a parallel case statement, and impose variable conditions and a condition of redundancy-freeness to obtain uniqueness of normal forms. In Section 4 we introduce the category of constrained environments, where objects are environments (type assignments) equipped with equational constraints. This will serve as the underlying category of our Grothendieck topology which is defined in Section 5. There we also introduce the category of sheaves for this topology and its bicartesian closed structure. This yields a canonical interpretation of the syntax in the category of sheaves and in Section 6 we show how to invert this interpretation and obtain normal forms.

2 Syntax

We follow the treatment of sums in natural deduction, as in [GLT89, pp 80-81]. For ease of presentation, we restrict ourselves to one base type.

Types are given by the grammar

$$A ::= o \mid A \Rightarrow A \mid A \times A \mid \top \mid A + A$$

Terms are given by

$$e \quad ::= \quad x \mid \lambda x.e \mid e\,e \mid \langle e,e\rangle \mid \pi_0(e) \mid \pi_1(e) \mid \langle\rangle \mid \\ \iota_0(e) \mid \iota_1(e) \mid \delta\ (x.e)\ (x.e)\ e$$

The Case term $\delta\ (x_0.e_0)\ (x_1.e_1)\ e_2$ simultaneously binds x_0 in e_0 and x_1 in e_1.

A *type environment* Γ is a finite function from variables to types. The typing judgement $\Gamma \vdash e : A$ meaning *e has type A in type environment* Γ is defined in the obvious way. For example, the rule for Case is:

$$\frac{(\Gamma, x_i : A_i \vdash e_i : C)_{i\in\{0,1\}} \qquad \Gamma \vdash e : A_0 + A_1}{\Gamma \vdash \delta\ (x_0.e_0)\ (x_1.e_1)\ e : C}$$

Definition 2.1 *Equality between terms in environment* Γ, denoted $\Gamma \vdash - = - : A$, is the least (typed) congruence generated by the following rules (omitting types to improve readability):

(β)	$(\lambda x.e_0)e_1 = e_0[e_1/x]$
(η)	$e = \lambda x.e\,x, \quad \text{if } x \notin \mathrm{FV}(e)$
Proj_i	$\pi_i(\langle e_0, e_1\rangle) = e_i$
SP	$e = \langle \pi_0(e), \pi_1(e)\rangle$
Unit	$e = \langle\rangle$
In_i	$\delta\ (x_0.e_0)\ (x_1.e_1)\ \iota_i(e_2) = e_i[e_2/x_i]$
Coprod	$\delta\ (x_0.\iota_0(x_0))\ (x_1.\iota_1(x_1))\ e = e$
Distrib	$e\,(\delta\ (x_0.e_0)\ (x_1.e_1)\ e_2) =$ $\delta\ (x_0.e\,e_0)\ (x_1.e\,e_1)\ e_2$ if $x_0, x_1 \notin \mathrm{FV}(e)$

We will refer to this equational theory as BiCCC. The key categorical axiom (Coprod) is dual to (SP) and guarantees uniqueness of the co-pairing arrow out of a coproduct. BiCCC entails all the usual commutative conversions for sums, [GLT89], pp. 80-81.

It can be shown (cf. [LS86, CDS97]) that the free almost bicartesian closed category $\mathcal{B}_0$ over one base object o can be obtained as the category whose objects are type environments and where a morphism from $\Gamma = x_1 : A_1, \ldots, x_m : A_m$ to $\Delta = y_1 : B_1, \ldots, y_n : B_n$ is a sequence of terms $(e_1, \ldots, e_n)$, modulo BiCCC equality, where $\Gamma \vdash e_i : B_i$. *Freeness* means that for each BiCCC $\mathcal{B}$ and object $[\![o]\!] \in \mathcal{B}$ we have a unique structure- and equation-preserving interpretation functor $[\![-]\!] : \mathcal{B}_0 \to \mathcal{B}$.

3 Normal Forms

Normal forms are defined simultaneously with *pure normal forms* and *neutral terms*. Normal (and pure normal) forms are not genuine terms, but defined inductively by the clauses below. If Γ is a type environment we write $\Gamma \vdash_{\mathrm{NF}} t : A$, resp. $\Gamma \vdash_{\mathrm{PNF}} t : A$, resp. $\Gamma \vdash_{\mathrm{NE}} t : A$ to mean that expression t is a normal form, resp. pure normal form, resp. neutral term of type A. We write $\mathrm{FV}(t)$ for the set of free variables occurring in t. We write $\mathrm{Guards}(t)$ for the set of *guards* of a normal form t; this will be defined below as part of the rule for forming normal forms.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\mathrm{NE}} x : \Gamma(x)} \qquad \frac{\Gamma \vdash_{\mathrm{NE}} s : o}{\Gamma \vdash_{\mathrm{PNF}} s : o}$$

$$\Gamma \vdash_{\mathrm{PNF}} \langle\rangle : \top$$

$$\frac{\Gamma \vdash_{\mathrm{PNF}} t_0 : A_0 \qquad \Gamma \vdash_{\mathrm{PNF}} t_1 : A_1}{\Gamma \vdash_{\mathrm{PNF}} \langle t_0, t_1\rangle : A_0 \times A_1}$$

$$\frac{\Gamma \vdash_{\mathrm{NE}} t : A_0 \times A_1}{\Gamma \vdash_{\mathrm{NE}} \pi_i(t) : A_i} \qquad i \in \{0,1\}$$

$$\frac{\Gamma \vdash_{\mathrm{PNF}} t : A_i}{\Gamma \vdash_{\mathrm{PNF}} \iota_i(t) : A_0 + A_1} \qquad i \in \{0,1\}$$

$$\frac{\Gamma \vdash_{\mathrm{NE}} s : A \Rightarrow B \qquad \Gamma \vdash_{\mathrm{PNF}} t : A}{\Gamma \vdash_{\mathrm{NE}} st : B}$$

$$\frac{\Gamma, x{:}A \vdash_{\mathrm{NF}} t : B}{\Gamma \vdash_{\mathrm{PNF}} \lambda x.t : A \Rightarrow B}$$

where in the last rule we have the variable condition that $x \in \mathrm{FV}(s)$ for each $s \in \mathrm{Guards}(t)$.

We have two rules for forming normal forms:

(a) $$\frac{\Gamma \vdash_{\mathrm{PNF}} t : A}{\Gamma \vdash_{\mathrm{NF}} t : A} \quad \text{and} \quad \mathrm{Guards}(t) = \emptyset$$

(b) Let $M = \{s_1, \ldots, s_n\}$ be a nonempty finite set of neutral terms (so we assume the s_i are pairwise distinct). For each $f : M \to \{0,1\}$ we use the abbreviation $\Gamma_f = \Gamma, x_1{:}A^1_{f(s_1)}, \ldots, x_n{:}A^n_{f(s_n)}$. Define

$$\frac{(\Gamma \vdash_{\mathrm{NE}} s_i : A^i_0 + A^i_1)_{i\in\{1,\ldots,n\}} \qquad (\Gamma_f \vdash_{\mathrm{NF}} t_f : C)_{f:M\to\{0,1\}}}{\Gamma \vdash_{\mathrm{NF}} \mathcal{C}(M, (x_1 \cdots x_n.t_f)_f) : C}$$

and $$\mathrm{Guards}(\mathcal{C}(M, (x_1 \cdots x_n.t_f)_f)) = M$$

where $(t_f)_{f:M\to\{0,1\}}$ is a family of normal forms satisfying the following two side conditions:

Variable-condition: for each $s \in \mathrm{Guards}(t_f)$ we have $\{x_1, \ldots, x_n\} \cap \mathrm{FV}(s) \neq \emptyset$.

Redundancy-freeness: The family $(t_f)_f$ is not redundant at any $s_i \in M$, where $(t_f)_f$ is redundant at s_i whenever for all $g : M \setminus \{s_i\} \to \{0,1\}$, $t_{g[s_i\mapsto 0]}$ and $t_{g[s_i\mapsto 1]}$ are equal and neither contains the variable x_i.

The variables $x_1, \ldots, x_n$ become bound in the $\mathcal{C}$-construct. For brevity we shall often use the al-

ternative notation $\mathcal{C}(M,(t'_f)_f)$, where the t'_f range over abstractions $x_1,\ldots,x_n.t_f$.

The idea is that $\mathcal{C}$ performs a simultaneous case split over all the "guards". For example, $t_{f[s\mapsto 0]}$ corresponds to a branch to be taken when s is of the form $\iota_0(x)$.

Example 3.1 The following examples show how the side-conditions ensure uniqueness of normal forms as computed by nf in Section 1. For simplicity let the variables z (possibly with indices) in the examples below have type o, so that they are normal terms.

1. The normal form of $\lambda w.\delta\ (x_1.z_0)\ (x_1.z_1)\ y$ will be $\mathcal{C}(\{y\},(x_1.t_f)_f)$ where $t_{[y\mapsto i]} = \lambda w.z_i$. Note that the expression $\lambda w.\mathcal{C}(\{y\},(x_1.t_f)_f)$, where $t_{[y\mapsto i]} = z_i$, violates the side condition for (pure) normal forms of λ-form.

2. The normal form of the term

$$\begin{array}{l}\delta\ (x_1.\delta\ (x_2.z_{00})\ (x_2.z_{01})\ y_2)\\ \quad(x_1.\delta\ (x_2.z_{10})\ (x_2.z_{11})\ y_2)\\ \quad y_1\end{array}$$

will be

$$\mathcal{C}(\{y_1,y_2\},(x_1x_2.t_f)_f)$$

where $t_{[y_1\mapsto i,y_2\mapsto j]} = z_{ij}$. Note that the expression $\mathcal{C}(\{y_1\},(x_1.\mathcal{C}(\{y_2\},(x_2.t_{f_1\cup f_2})_{f_2})_{f_1}))$ is not a normal form since it violates the variable-condition: x_1 is not free in the guard y_2 of the normal form $\mathcal{C}(\{y_2\},(x_2.t_{f_1\cup f_2})_{f_2})$.

3. The normal form of $\delta\ (x.z)\ (x.z)\ y$ will be z. Note that $\mathcal{C}(\{y\},(x.z)_f)$ is not a normal form as $(x.z)_f$ is redundant at y.

4. Note however, that the normal form of $\delta\ (z.z)\ (z.z)\ y$ will be $\mathcal{C}(\{y\},(z.z)_f)$ which is not redundant at y because of the variable condition in the definition of redundancy.

Definition 3.2 The function d mapping $\Gamma\vdash_X t:A$ with $X\in\{\mathrm{NF},\mathrm{PNF},\mathrm{NE}\}$ to terms $\Gamma\vdash \mathsf{d}(t):A$ is defined in the following way:

- d commutes with all the term formers except $\mathcal{C}$ (in particular, preserves variables).
- $\mathsf{d}(\mathcal{C}(M\cup\{s\},(t_f)_f)) = \delta\ (x_0.e_0)\ (x_1.e_1)\ \mathsf{d}(s)$, where $e_i = \mathsf{d}(\mathcal{C}(M,(t_{g[s\mapsto i]})_g))$.

It is easy to see that up to BiCCC equality this does not depend on the choice of the witnessing term e_Γ and on the order of the guards.

4 Neutral constrained environments

Like Dougherty and Subrahmanyam [DS95] and Fiore and Simpson [FS99] we need to supply our type environments with constraints. These will be the objects of a category of constrained environments $\mathcal{N}$, where the morphisms will be injective renamings. The constraints are of the form $s=\iota_i(x_i)$ and express which branch a certain guard s takes. This is the idea behind our Grothendieck topology on $\mathcal{N}$: a "covering" expresses case-splitting. This use of Grothendieck topologies is closely related to [FS99] where they were used for proving a definability result for a language with coproducts.

Definition 4.1 A *neutral constrained environment*, environment for short, is a pair $\Gamma\,|\,\Xi$ where Γ is a type environment and Ξ is a set of constraints of the form $s=\iota_0(x_0)$ or $s=\iota_1(x_1)$ where $\Gamma\vdash_{\mathrm{NE}} s:A_0+A_1$ and $x_0:A_0$ (resp. $x_1:A_1$) is contained in Γ and moreover,

- no two distinct constraints involve the same neutral term, for example, Ξ cannot contain $s{=}\iota_0(x_0)$ and $s{=}\iota_1(x_1)$
- no two distinct constraints refer to the same variable, for example, Ξ cannot contain $s=\iota_0(x_0)$ and $s'=\iota_0(x_0)$ unless s and s' are identical.

A *morphism* from environment $\Delta\,|\,\Psi$ to environment $\Gamma\,|\,\Xi$ is given by an *injective* function $\sigma:\mathrm{dom}(\Gamma)\to\mathrm{dom}(\Delta)$ satisfying $\Delta(\sigma(x))=\Gamma(x)$ and $\sigma(s){=}\iota_i(\sigma(x))$ is in Ψ for each constraint $s{=}\iota_i(x)$ in Ξ. In this way the environments form a category $\mathcal{N}$ in which composition is composition of functions.

If Δ extends Γ and Ψ extends Ξ then the inclusion $\sigma:\mathrm{dom}(\Gamma)\hookrightarrow\mathrm{dom}(\Delta)$ defines a morphism from $\Delta\,|\,\Psi$ to $\Gamma\,|\,\Xi$ which we call a *projection*.

We are interested in studying equality of terms relative to a neutral constrained environment. The following definition is due to [DS95].

Definition 4.2 Let $\Gamma\,|\,\Xi$ be an environment and $\vec{d}$ be a list of dummy terms of the same length as Ξ and of appropriate (to be explained) type. A (variable-binding) type environment $C^{\Gamma\,|\,\Xi}_{\vec{d}}[\,]$ is defined as follows.

$$C^{\Gamma\,|\,\emptyset}[\,] = [\,]$$

$$C^{\Gamma,x_0:A_0\,|\,\Xi,s=\iota_0(x_0)}_{\vec{d},d_1}[\,] = \delta\ (x_0.C^{\Gamma\,|\,\Xi}_{\vec{d}}[\,])\ (x_1.d_1x_1)\ \mathsf{d}(s)$$

$$C^{\Gamma,x_1:A_1\,|\,\Xi,s=\iota_1(x_1)}_{\vec{d},d_0}[\,] = \delta\ (x_0.d_0x_0)\ (x_1.C^{\Gamma\,|\,\Xi}_{\vec{d}}[\,])\ \mathsf{d}(s)$$

Note that $C^{\Gamma\,|\,\Xi}_{\vec{d}}[e]$ binds all variables mentioned in Ξ.

Given two terms $\Gamma \vdash e_1 : C$ and $\Gamma \vdash e_2 : C$ we write $\Gamma \mid \Xi \vdash e_1 = e_2 : C$ to mean that

$$\Gamma' \vdash C_{\vec{d}}^{\Gamma \mid \Xi}[e_1] = C_{\vec{d}}^{\Gamma \mid \Xi}[e_2] : C$$

in the theory BiCCC for all appropriate Γ' and $\vec{d}$. Here $\vec{d}$ must be chosen such that the terms $C_{\vec{d}}^{\Gamma \mid \Xi}[e_i]$ are type correct and Γ' is obtained from Γ by removing the variables mentioned in Ξ and possibly adding any extra free variables occurring in the dummy terms $\vec{d}$.

Remark 4.3 Note that ordinary type environments have no constraints but it follows immediately from the above definition that $\Gamma|\emptyset \vdash e_1 = e_2$ implies $\Gamma \vdash e_1 = e_2$.

5 Sheaves over environments

We consider the functor category $\widehat{\mathcal{N}} \stackrel{\text{def}}{=} \mathit{Sets}^{\mathcal{N}^{\mathrm{op}}}$ of presheaves and natural transformations between them. We recall the following definitions of the structure of $\widehat{\mathcal{N}}$. A *presheaf* is given by a family of sets $F_{\Gamma \mid \Xi}$ indexed by environments and for each morphism $\sigma : \Delta \mid \Psi \to \Gamma \mid \Xi$ a function $F_\sigma : F_{\Gamma \mid \Xi} \to F_{\Delta \mid \Psi}$ such that $F_1 = 1$ and $F_{\sigma \circ \tau} = F_\tau \circ F_\sigma$. If $a \in F_{\Gamma \mid \Xi}$ we may write $a\restriction_{\Delta \mid \Psi}$ for $F_\sigma(a)$ in case σ is clear from the context. This notation will in particular be used when σ is a projection.

A *natural transformation* from presheaf F to presheaf G is given by a family $g_{\Gamma \mid \Xi}$ of maps $g_{\Gamma \mid \Xi} : F_{\Gamma \mid \Xi} \to G_{\Gamma \mid \Xi}$ such that $G_\sigma \circ g_{\Gamma \mid \Xi} = g_{\Delta \mid \Psi} \circ F_\sigma$ (*naturality*). If $a \in F_{\Gamma \mid \Xi}$ we may write $g(a)$ for $g_{\Gamma \mid \Xi}(a)$. Naturality then reads $g(a)\restriction_{\Delta \mid \Psi} = g(a\restriction_{\Gamma \mid \Xi})$.

As any category of presheaves, the category $\widehat{\mathcal{N}}$ is bicartesian closed, that is, supports the interpretation of the type formers $\top, \times, \Rightarrow, +$, (and $\bot$). If we denote the interpreting presheaves with the same symbols thus writing e.g. $F \Rightarrow G$ for the function space of presheaves, we have the following explicit constructions of the type formers in $\mathit{Sets}^{\mathcal{N}^{\mathrm{op}}}$:

$$\begin{array}{lcl} \top_{\Gamma \mid \Xi} & = & \{\langle\rangle\} \\ (F \times G)_{\Gamma \mid \Xi} & = & F_{\Gamma \mid \Xi} \times G_{\Gamma \mid \Xi} \\ (F + G)_{\Gamma \mid \Xi} & = & F_{\Gamma \mid \Xi} + G_{\Gamma \mid \Xi} \\ (F \Rightarrow G)_{\Gamma \mid \Xi} & = & \widehat{\mathcal{N}}(\mathcal{N}(-, \Gamma \mid \Xi) \times F\ G) \end{array}$$

However, as we mentioned in the introduction, we are not able to obtain normal forms by inverting this presheaf interpretation. Instead we shall consider the interpretation of terms in the category of sheaves over a certain topology, and show that this can be inverted.

Recall that the basis of a *Grothendieck topology* is a collection of *basic coverings*, satisfying the axioms of identity, transitivity, and stability [MM92, p.111]. A covering of an object $\Gamma \mid \Xi$ in $\mathcal{N}$ is here a family of arrows with codomain $\Gamma \mid \Xi$. Since the category $\mathcal{N}$ does not have pullbacks in general, we use a modified axiom of stability [MM92, p.156]. Moreover, like [FS99] we only require that the identity is a singleton covering, not that all isomorphisms are coverings.

Definition 5.1 The basis K for a Grothendieck topology on $\mathcal{N}$ is inductively generated by the following clauses:

- The identity covering containing only the arrow $1_{\Gamma \mid \Xi}$ is a basic covering of $\Gamma \mid \Xi$.
- If $\Gamma \vdash_{\mathrm{NE}} s : A_0 + A_1$ and s is not mentioned in Ξ, and if the family of projections from $(\Gamma_i \mid \Xi_i)_i$ forms a basic covering of $\Gamma, x_0 : A_0 | \Xi, s = \iota_0(x_0)$ and the family of projections from $(\Gamma_j \mid \Xi_j)_j$ forms a basic covering of $\Gamma, x_1 : A_1 | \Xi, s = \iota_1(x_1)$, then the disjoint union of the projections from $(\Gamma_i \mid \Xi_i)_i$ and $(\Gamma_j \mid \Xi_j)_j$ forms a basic covering of $\Gamma \mid \Xi$.

The general concept of sheaves for Grothendieck topologies need not be presented, since it here specialises to the following rather digestible definition:

Proposition 5.2 *A presheaf F is a* sheaf *for K iff whenever $\Gamma \mid \Xi$ is covered by $\Gamma, x_0{:}A_0 \mid \Xi, s{=}\iota_0(x_0)$ and $\Gamma, x_1{:}A_1 \mid \Xi, s{=}\iota_1(x_1)$, that is, $\Gamma \vdash_{\mathrm{NE}} s : A_0 + A_1$ and*

$$\begin{array}{rcl} f_0 & \in & F_{\Gamma, x_0:A_0 \mid \Xi, s=\iota_0(x_0)} \\ f_1 & \in & F_{\Gamma, x_1:A_1 \mid \Xi, s=\iota_1(x_1)} \end{array}$$

then there exists a unique $f \in F_{\Gamma \mid \Xi}$ *(called* pasting*) such that*

$$\begin{array}{rcl} f\restriction_{\Gamma, x_0:A_0 \mid \Xi, s=\iota_0(x_0)} & = & f_0 \\ f\restriction_{\Gamma, x_1:A_1 \mid \Xi, s=\iota_1(x_1)} & = & f_1 \end{array}$$

The following result follows from general properties of Grothendieck topologies and will therefore not be proved, see [MM92] for an exposition.

Proposition 5.3

1. *The terminal object in $\widehat{\mathcal{N}}$ is a sheaf,*
2. *if F, G are sheaves so is $F \times G$ (cartesian product),*
3. *if G is a sheaf and F is a presheaf then $F \Rightarrow G$ is a sheaf (function space)*
4. *for each presheaf F there exists a sheaf aF (the* associated sheaf *or* sheafification*) and a natural transformation $\eta : F \to aF$ such that whenever G is a sheaf and $f : F \to G$ then there exists a unique $f^\sharp : aF \to G$ with $f^\sharp \circ \eta = f$. In other words, the sheaves form a reflective subcategory of $\mathcal{N}$,*
5. *The sheafification functor a preserves binary products.*

6. *if F, G are sheaves the coproduct $F + G$ is in general not a sheaf, but $a(F + G)$ is the coproduct of F and G in the subcategory of sheaves.*

7. *if $u, v : F \to G$ and F, G are sheaves then the equaliser of u and v is a sheaf.*

We write $\Gamma \mid \Xi \vdash_{\mathrm{NF}} t : A$ to mean that $\Gamma \vdash_{\mathrm{NF}} t : A$ and, moreover, none of the neutral terms mentioned in Ξ is contained in $\mathrm{Guards}(t)$. Intuitively, this is because no case split is ever needed for a guard whose value is already known through the environment. Note that there is no need to define $\Gamma \mid \Xi \vdash_{\mathrm{NE}} t : A$ and $\Gamma \mid \Xi \vdash_{\mathrm{PNF}} t : A$, since all guards inside neutral and pure normal terms include variables bound by λ's. Hence the constraints in Ξ cannot affect t.

For a type A we define the presheaves $\mathrm{NF}(A)$, $\mathrm{PNF}(A), \mathrm{NE}(A), \mathrm{Term}(A)$ as follows:

$$\begin{aligned}
\mathrm{NF}(A)_{\Gamma \mid \Xi} &= \{t \mid \Gamma \mid \Xi \vdash_{\mathrm{NF}} t : A\} \\
\mathrm{PNF}(A)_{\Gamma \mid \Xi} &= \{t \mid \Gamma \vdash_{\mathrm{PNF}} t : A\} \\
\mathrm{NE}(A)_{\Gamma \mid \Xi} &= \{t \mid \Gamma \vdash_{\mathrm{NE}} s : A\} \\
\mathrm{Term}(A)_{\Gamma \mid \Xi} &= \{t \mid \Gamma \mid \Xi \vdash t : A\}/{\sim_{\vdash}}
\end{aligned}$$

where $t \sim_{\vdash} t'$ stands for $\Gamma \mid \Xi \vdash t = t' : A$.

If $\sigma : \Delta \mid \Psi \to \Gamma \mid \Xi$ and $\Gamma \mid \Xi \vdash_{\mathrm{NE}} t : A$ then $\mathrm{NE}(A)_\sigma(t) \in \mathrm{NE}(A)_{\Delta \mid \Psi}$ is defined by replacing each free variable x in t by $\sigma(x)$. The morphism parts Term_σ and PNF_σ are defined analogously.

If $t \in \mathrm{NF}_{\Gamma \mid \Xi}(A)$ then $\mathrm{NF}_\sigma(t)$ is defined by first replacing each free variable x in t by $\sigma(x)$ and then plugging in all the constraints mentioned in Ψ by repeatedly performing the following atomic restriction operation (an analogous operation appears in Ghani's thesis [Gh95a] under the name "first and second residue").

Definition 5.4 Let $t \in \mathrm{NF}(C)_{\Gamma \mid \Xi}$ and $\Gamma \vdash_{\mathrm{NE}} s : A_0 + A_1$. Then we define the restriction $t[s{:=}\iota_i(x_i)]$ of t to $\Gamma, x_i : A_i \mid \Xi, s{=}\iota_i(x_i)$ (along the projections) as follows.

$$\begin{aligned}
t[s{:=}\iota_i(x)] &= t, \text{ if } s \notin \mathrm{Guards}(t) \\
\mathcal{C}(M \cup \{s\}, (t_f)_f)[s{:=}\iota_i(x_i)] &= \mathcal{C}^{\mathrm{nf}}(M, (t_{g[s \to i]})_g)
\end{aligned}$$

where $\mathcal{C}^{\mathrm{nf}}$ computes a normal form to be defined below. Note that we cannot simply replace $\mathcal{C}^{\mathrm{nf}}$ by $\mathcal{C}$ because the set of guards can become empty upon plugging in a constraint, new redundancies may be created, and the variable conditions may be violated. We define $\mathcal{C}^{\mathrm{nf}}(\emptyset, \{t\})$ to be t and $\mathcal{C}(M \cup \{s\}, (t_f)_f)$ to be $\delta^{\mathrm{nf}}\ (x_0.\mathcal{C}^{\mathrm{nf}}(M, (t_{f[s \mapsto 0]})_f))\ (x_1.\mathcal{C}^{\mathrm{nf}}(M, (t_{f[s \mapsto 1]})))\ s$.

To compute $\delta^{\mathrm{nf}}\ (x_0.t_0)\ (x_1.t_1)\ s$ we first check whether t_i depend on x_i and are different (see the definition of redundancy). If not, we return $t_0 (= t_1)$, or otherwise, we return $\mathcal{C}(\{s\} \cup M_0 \cup M_1, t_g)$, where

$$M_i = \{s_i \in \mathrm{Guards}(t_i) | x_i \notin \mathrm{FV}(s_i)\}$$

for $i = 0, 1$, and the family t_g is adjusted accordingly.

Proposition 5.5 d *defines natural transformations* $\mathrm{NF}(A) \to \mathrm{Term}(A)$, $\mathrm{PNF}(A) \to \mathrm{Term}(A)$, $\mathrm{NE}(A) \to \mathrm{Term}(A)$.

If $f : \mathcal{B}(\Delta, \Gamma)$ is a morphism in the free BiCCC $\mathcal{B}$, that is, a sequence of terms in type environment Δ, then $[t] \mapsto [ft]$ defines a natural transformation $\mathrm{Term}(f) : \mathrm{Term}(\Delta) \to \mathrm{Term}(\Gamma)$. This makes $\mathrm{Term}(-)$ a functor from $\mathcal{B}$ to $\widehat{\mathcal{N}}$ preserving $\top$ and cartesian products.

Proposition 5.6 *The presheaf* $\mathrm{Term}(A)$ *is a sheaf.*

Proposition 5.7 *The presheaf* $\mathrm{NF}(A)$ *is a sheaf and is isomorphic to the sheafification* $a(\mathrm{PNF}(A))$ *of* $\mathrm{PNF}(A)$ *with the embedding* $\eta : \mathrm{PNF}(A) \to \mathrm{NF}(A)$ *given by* $\eta_{\Gamma \mid \Xi}(t) = t$.

If $\Gamma \vdash s : A_0 + A_1$, then the pasting of two normal forms $t_i \in \mathrm{NF}(A)_{\Gamma, x_i : A_i \mid \Xi, s = \iota_i(x_i)}$ is the normal form $\delta^{\mathrm{nf}}(x_0.t_0)(x_1.t_1)s \in \mathrm{NF}(A)_{\Gamma \mid \Xi}$.

Let us write $Sh(\mathcal{N})$ for the full subcategory of $\widehat{\mathcal{N}}$ consisting of the sheaves. We know from Prop. 5.3 that $Sh(\mathcal{N})$ is a BiCCC. Since the category $\mathcal{B}_0$ of sequences of types and terms is a free BiCCC there is a unique interpretation functor $[\![-]\!] : \mathcal{B}_0 \to Sh(\mathcal{N})$, determined by

$$[\![o]\!] = \mathrm{NF}(o)$$

Concretely, this functor is given by defining a canonical BiCCC structure on $Sh(\mathcal{N})$.

6 Inverting the interpretation function

We will now define natural transformations

$$\begin{aligned}
\mathsf{q}^A &: [\![A]\!] \to \mathrm{NF}(A) \\
\mathsf{u}^A &: \mathrm{NE}(A) \to [\![A]\!]
\end{aligned}$$

in such a way that for a term $\Gamma \vdash e : A$,

$$\mathsf{nf}(e) \stackrel{\mathrm{def}}{=} \mathsf{q}_\Gamma^A([\![e]\!](\mathsf{u}_\Gamma^\Gamma(1_\Gamma)))$$

will satisfy NF1:

- $\mathsf{q}^o : \mathrm{NF}(o) \to \mathrm{NF}(o)$ is the identity function.

  $\mathsf{u}^o : \mathrm{NE}(o) \to \mathrm{NF}(o)$ is the injection from neutral terms to normal terms given by the obvious term-formation rules.

- $\mathsf{q}^\top : \top \to \mathrm{NF}(\top)$ is the constant function returning the normal form $\langle\rangle$.

  $\mathsf{u}^\top : \mathrm{NE}(\top) \to \top$ is the constant function returning the element $\langle\rangle \in \top$. (As before we use the same signs for corresponding syntactic and semantic notions.)

- $\mathsf{q}^{A_0 \times A_1} = \mathrm{pair}^{\mathrm{nf}} \circ (\mathsf{q}^{A_0} \times \mathsf{q}^{A_1})$ where $\mathrm{pair}^{\mathrm{nf}} : \mathrm{NF}(A_0) \times \mathrm{NF}(A_1) \to \mathrm{NF}(A_0 \times A_1)$ is the unique map satisfying $\mathrm{pair}^{\mathrm{nf}}(t_1, t_2) = \langle t_1, t_2 \rangle$ for *pure* normal forms t_1, t_2. This map exists by Proposition 5.7 and the fact that a preserves products.

$$\mathsf{u}^{A_0 \times A_1}_{\Gamma \mid \Xi}(s) = \langle \mathsf{u}^{A_0}_{\Gamma \mid \Xi}(\pi_0(s)), \mathsf{u}^{A_1}_{\Gamma \mid \Xi}(\pi_1(s)) \rangle$$

- Let $\theta \in [\![A \Rightarrow B]\!]_{\Gamma \mid \Xi} = \widehat{\mathcal{N}}(\mathcal{N}(-, \Gamma \mid \Xi) \times [\![A]\!], [\![B]\!])$. Then

$$\mathsf{q}^{A \Rightarrow B}_{\Gamma \mid \Xi}(\theta) = \lambda^{\mathrm{nf}} x. \mathsf{q}^{B}_{\Gamma, x:A \mid \Xi}(\theta(\sigma, \mathsf{u}^{A}_{\Gamma, x:A \mid \Xi}(x)))$$

where σ is the projection from $\Gamma, x : A | \Xi$ to $\Gamma \mid \Xi$. Here $\lambda^{\mathrm{nf}} x. \mathcal{C}(M, (x_1 \ldots x_n . t_f)_f)$ is obtained by dividing M into two sets, M_0 which contains the guards which do not depend on x, and M_1, which contains the guards which do. Then we return

$$\mathcal{C}(M_0, (x_1 \ldots x_{n_0} . \lambda x. \mathcal{C}^{\mathrm{nf}}(M_1, (x_1 \ldots x_{n_1} . t_{f_0 \cup f_1})_{f_1}))_{f_0})$$

Compare also example 1 in 3.1.

Let $s \in \mathrm{NE}(A \Rightarrow B)_{\Gamma \mid \Xi}$. Then $\mathsf{u}^{A \Rightarrow B}_{\Gamma \mid \Xi}(s) \in [\![A \Rightarrow B]\!]_{\Gamma \mid \Xi}$ is defined by

$$(\mathsf{u}^{A \Rightarrow B}_{\Gamma \mid \Xi}(s))_{\Delta \mid \Psi}(\sigma, a) =$$
$$\mathsf{u}^{B}_{\Delta \mid \Psi}(\mathrm{NE}_{\sigma}(s)(\mathsf{q}^{A}_{\Delta \mid \Psi}(a))) \in [\![B]\!]_{\Delta \mid \Psi}$$

where $\sigma \in \mathcal{N}(\Delta \mid \Psi, \Gamma \mid \Xi)$ and $a \in [\![A]\!]_{\Delta \mid \Psi}$.

- $\mathsf{q}^{A_0 + A_1}$ is the unique map (arising from the coproduct property of $[\![A_0 + A_1]\!]$) satisfying

$$\mathsf{q}^{A_0 + A_1}(\iota^{\mathrm{sh}}_0(a)) = \iota^{\mathrm{nf}}_0(\mathsf{q}^{A_0}(a))$$
$$\mathsf{q}^{A_0 + A_1}(\iota^{\mathrm{sh}}_1(b)) = \iota^{\mathrm{nf}}_1(\mathsf{q}^{A_1}(b))$$

Here $\iota^{\mathrm{sh}}_0, \iota^{\mathrm{sh}}_1$ are the coproduct injections in $Sh(\mathcal{N})$ and $\iota^{\mathrm{nf}}_0 : \mathrm{NF}(A_0) \to \mathrm{NF}(A_0 + A_1)$ is the unique map satisfying $\iota^{\mathrm{nf}}_0(t) = \iota_0(t)$ for *pure* normal form $t : A_0$. Similarly for ι^{nf}_1.

To construct

$$\mathsf{u}^{A_0 + A_1} \in \mathrm{NE}(A_0 + A_1) \to [\![A_0 + A_1]\!]$$

consider $s \in \mathrm{NE}(A_0 + A_1)_{\Gamma \mid \Xi}$: either $s = \iota_0(x) \in \Xi$ in which case we put $f_{\Gamma \mid \Xi}(s) = \iota^{\mathrm{sh}}_0(\mathsf{u}^{A_0}_{\Gamma \mid \Xi}(x))$, or $s = \iota_1(y) \in \Xi$ and we put $f_{\Gamma \mid \Xi}(s) = \iota^{\mathrm{sh}}_1(\mathsf{u}^{A_1}_{\Gamma \mid \Xi}(y))$, or s is not mentioned in Ξ in which case we define $f_{\Gamma \mid \Xi}(s)$ as the unique pasting of

$$a_0 \stackrel{\mathrm{def}}{=} \iota^{\mathrm{sh}}_0(\mathsf{u}^{A_0}_{\Gamma, x:A_0 \mid \Xi, s=\iota_0(x)}(x))$$
$$a_1 \stackrel{\mathrm{def}}{=} \iota^{\mathrm{sh}}_1(\mathsf{u}^{A_1}_{\Gamma, x:A_1 \mid \Xi, s=\iota_1(x)}(x))$$

It follows by straightforward calculations that all these are indeed natural transformations.

Proposition 6.1 *In order to establish NF1, that is,* $e = \mathsf{d}(\mathsf{q}^{A}([\![e]\!](\mathsf{u}(1_\Gamma))))$ *for* $\Gamma \vdash e : A$ *we define a family of subsheaves* $R^{A}_{\Gamma \mid \Xi} \subseteq [\![A]\!]_{\Gamma \mid \Xi} \times \mathrm{Term}(A)_{\Gamma \mid \Xi}$, *such that*

(i) *For all* $a \in [\![A]\!]_{\Gamma \mid \Xi}$ *and* $\Gamma \vdash e : A$:

$$a R^{A}_{\Gamma \mid \Xi} e \Rightarrow \Gamma \mid \Xi \vdash \mathsf{d}(\mathsf{q}^{A}_{\Gamma \mid \Xi}(a)) = e$$

(ii) *For all* $s \in \mathrm{NE}(A)_{\Gamma \mid \Xi}$

$$\mathsf{u}^{A}_{\Gamma \mid \Xi}(s) R^{A}_{\Gamma \mid \Xi} \mathsf{d}(s)$$

We can extend R to type environments by letting $(a_1, \ldots, a_n)\ R^{\Gamma}_{\Gamma \mid \Xi}\ (f_1, \ldots, f_n)$ iff $a_i\ R^{A_i}_{\Gamma \mid \Xi}\ f_i$ for $1 \leq i \leq n$, where $\Gamma = x_1 : A_1, \ldots x_n : A_n$. Similarly, we can extend q and u to type environments as well.

Proposition 6.2 (Logical Relations Lemma) *If* $\Gamma \vdash e : C$ *and* $\vec{a}\ R^{\Gamma}_{\Gamma \mid \Xi}\ \vec{f}$ *then*

$$[\![e]\!](\vec{a})\ R^{C}_{\Gamma \mid \Xi}\ e[\vec{f}/\vec{x}],$$

where $\vec{x}$ *are the variables in* Γ.

Theorem 6.3 *The equational theory BiCCC is decidable.*

Proof. The above shows that the normalisation function nf satisfies NF1, because by (ii) and $\mathsf{d}(1_\Gamma) = 1_\Gamma$, we know that

$$\mathsf{u}^{\Gamma}_{\Gamma}(1_\Gamma) R^{\Gamma}_{\Gamma} 1_\Gamma$$

Hence by Proposition 6.2, we know that

$$[\![e]\!](\mathsf{u}^{\Gamma}_{\Gamma}(1_\Gamma)) R^{A}_{\Gamma} e$$

Hence, by (i) (cf. Remark 4.3)

$$\Gamma \vdash \mathsf{d}(\mathsf{nf}(e)) = \mathsf{d}(\mathsf{q}^{A}_{\Gamma}([\![e]\!](\mathsf{u}^{\Gamma}_{\Gamma}(1_\Gamma)))) = e$$

As we pointed out in the introduction NF2 holds automatically, and hence it follows that

$$\Gamma \vdash e_1 = e_2 \iff \mathsf{nf}(e_1) = \mathsf{nf}(e_2)$$

This yields a decision procedure since equality of normal forms is decidable. (Note that when writing the algorithm we represent the finite set of guards as a list or a tree, so that normal forms are only unique up to the ordering of the guards.) Furthermore, the interpretation in $Sh(\mathcal{N})$ as well as the definition of q, u are clearly algorithmic. In fact, the whole development can be formalised in extensional Martin-Löf type theory using standard methods for formalizing category theory in Martin-Löf type theory. This would be one way of demonstrating explicitly that all functions we construct by abstract mathematical means are computable. □

References

[AHS95] T. Altenkirch, M. Hofmann, T. Streicher, Categorical reconstruction of a reduction-free normalisation proof, *Proc. CTCS '95 Springer LNCS* **953**, 182–199.

[AHS96] T. Altenkirch, M. Hofmann, T. Streicher, Reduction- free normalisation for a polymorphic system, *11th Annual IEEE LICS Symposium*, 1996, 98-106.

[AC98] R. M. Amadio, P-L. Curien, *Selected Domains and Lambda Calculi*, Camb. Univ. Press, 1998.

[BBSSZ98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger and W. Zuber, Proof theory at work: Program development in the Minlog system, in: *Automated Deduction*, W. Bibel and P.H. Schmitt, eds., Vol. II, Kluwer 1998).

[BES98] U. Berger, M. Eberl, H. Schwichtenberg Normalization by evaluation, Prospects for hardware foundations (NADA), *Springer LNCS* **1546**, 1998, pp. 117–137.

[BS91] U. Berger and H. Schwichtenberg, An inverse to the evaluation functional for typed λ- calculus,*6th Annual IEEE LICS Symposium*, 1991, 203-211.

[CD97] T. Coquand and P. Dybjer, Intuitionistic Model Constructions and Normalization Proofs, *Math. Structures in Compter Science* **7**, 1997, 75-94.

[CDS97] D. Čubrić, P. Dybjer and P.J.Scott. Normalization and the Yoneda Embedding, *Math. Structures in Compter Science* **8**, No.2, 1997, 153-192.

[Da96] O. Danvy. Type-directed partial evaluation, POPL'96, ACM Press, 242-257.

[Da98] O. Danvy. Type-directed partial evaluation, Partial evaluation, Practice and Theory, *Proceedings of the 1998 DIKU Summer School, Springer LNCS* **1706**, 367-411.

[DiCo95] R. Di Cosmo. *Isomorphism of Types: from λ-calculus to information retrieval and language design*, Birkhäuser, 1995.

[Do93] D. Dougherty. Some lambda calculi with categorical sums and products, RTA5, *Springer LNCS* **690**, 1993, 135-151.

[DS95] D. Dougherty and R. Subrahmanyam, Equality between Functionals in the Presence of Coproducts, *Inf. & Comp.* (to appear). Prelim. version in: *10th Annual IEEE LICS Symposium*, 1995, 282-291.

[Fil01] A. Filinski, Normalization by Evaluation for the Computational Lambda-Calculus, to appear in the proceedings of TLCA 2001.

[FS99] M. Fiore and A. Simpson. Lambda Definability with Sums via Grothendieck Logical Relations, TLCA'99, *Springer LNCS* **1581**.

[Gh95a] N. Ghani, *Adjoint Rewriting*, PhD thesis, LFCS, Univ. of Edinburgh, Nov. 1995.

[Gh95b] N. Ghani, $\beta\eta$-equality for coproducts. *TLCA '95 Springer LNCS* **902**,1995,171-185.

[GLT89] J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, 1989.

[JGh95] C. B. Jay and N. Ghani, The virtues of eta expansion, *Journal of Functional Programing*, **5**, no.2, 1995,135-154.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*, Cambridge Studies in Advanced Mathematics **7**, Cambridge University Press, 1986.

[MM92] S. Mac Lane, I. Moerdijk. *Sheaves in Geometry and Logic*, Springer-Verlag, 1992.

[ML75] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part, *Logic Colloquium '73*, North-Holland, 1975.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages*, MIT Press, 1996.

Strong Normalisation in the π-Calculus

(Extended Abstract)

Nobuko Yoshida * Martin Berger * Kohei Honda *

Abstract

We introduce a typed π-calculus where strong normalisation is ensured by typability. Strong normalisation is a useful property in many computational contexts, including distributed systems. In spite of its simplicity, our type discipline captures a wide class of converging name-passing interactive behaviour. The proof of strong normalisability combines methods from typed λ-calculi and linear logic with process-theoretic reasoning. It is adaptable to systems involving state and other extensions. Strong normalisation is shown to have significant consequences, including finite axiomatisation of weak bisimilarity, a fully abstract embedding of the simply-typed λ-calculus with products and sums and basic liveness in interaction.

Strong normalisability has been extensively studied as a fundamental property in functional calculi, term rewriting and logical systems. This work is one of the first steps to extend theories and proof methods for strong normalisability to the context of name-passing processes.

1. Introduction

Background The formal study of types in programming languages and computational calculi has led to the understanding that types can ensure a wide range of desirable computational properties, ranging from error-free execution to logical specification of program behaviour. One important property in this context, widely found in typed λ-calculi, is *strong normalisation* (SN), which says that computation in programs necessarily terminates regardless of evaluation strategy. This is interesting from a logical viewpoint especially because, by the correspondence between proofs and programs, SN of certain λ-calculi implies consistency of the corresponding logical systems. For this reason functional calculi and logics have been the main focus in the study of strong normalisability so far.

The significance of SN is, however, not limited to this traditional setting. SN is also interesting in the context of communicating processes. As an example, consider a distributed client-server interaction: when a client requests some service, s/he may naturally wish the computation on the server's side to terminate and return an answer. SN is thus a basic requirement for, say, interaction between banks and their customers. As another example, the resource preservation guaranteed by SN has been one of the main reasons for Gunter and his colleagues to develop their typed programming language for active networks (PLAN) [15, 33] on the basis of a simply typed λ-calculus. Such languages require primitives for communication and concurrency. This suggests a systematic effort to extend the accumulated theories of functional SN types to the realm of interactivity is a worthwhile endeavour.

We are thus motivated to reposition and study strong normalisability in the context of process theory. In particular, is there a basic typed process calculus in which strongly normalising functional calculi are faithfully embeddable? By faithful, we mean that typability of the encoding automatically ensures strong normalisability of the source calculus. More ambitiously, can we obtain exact semantic correspondence, including full abstraction and full completeness? Obtaining affirmative answers to these questions would not be of mere theoretical interest: as typed λ-calculi offer a basic theory of procedure calls, a fundamental abstraction in programming, embeddability of SN functional calculi would capture interactive behaviour powerful enough to involve non-trivial procedural calls while maintaining SN. Exploration of strong normalisability in this broader context might also shed new light on typed functional computation itself.

The present work is a trial in this direction, introducing a typed π-calculus in which the first-order strongly normalising λ-calculi are fully abstractly embeddable. The type discipline simply adds the minimum form of causal chains to the system introduced in [8] where we established a fully abstract encoding of PCF. This small addition radi-

*Department of Mathematics and Computer Science, University of Leicester, UK. E-Mail: ny11@mcs.le.ac.uk. *Department of Computer Science, Queen Mary, University of London, E-Mail: {martinb, kohei}@dcs.qmw.ac.uk. Partially supported by EPSRC grant GR/N/37633.

0-7695-1281-X/01 $10.00 © 2001 IEEE

cally changes the class of typable process behaviour, turning possibly diverging computation into a strongly normalising one. As would be imagined by the embeddability of typed λ-calculi, the proof of SN is non-trivial, defying naive structural induction. We adapt methods developed for strongly normalising λ-calculi [6, 13, 37], combined with process-algebraic reasoning [8, 30, 32, 36, 40]. As far as we know, this is the first time a compositional principle for ensuring SN has been established for name passing processes with non-trivial use of replication. The proof method for SN is applicable to extensions of the presented formalism. In the following, we outline key technical ideas and relate our work to the existing literature.

The π-Calculus Following [8], we use an asynchronous variant of the π-calculus [10, 19]; computation in this calculus is generated by interaction between processes.

$$x(\vec{y}).P \,|\, \overline{x}\langle\vec{v}\rangle \longrightarrow P\{\vec{v}/\vec{y}\}$$

Here $\vec{y}$ denotes a potentially empty vector $y_1 ... y_n$, $|$ denotes parallel composition, $x(\vec{y}).P$ is input, and $\overline{x}\langle\vec{v}\rangle$ is asynchronous output. Operationally this reduction represents the consumption of an asynchronous message by a receptor. The idea extends to a receptor with replication

$$!x(\vec{y}).P \,|\, \overline{x}\langle\vec{v}\rangle \longrightarrow !x(\vec{y}).P \,|\, P\{\vec{v}/\vec{y}\},$$

where the replicated process remains in the configuration after reduction. As a simple example of a process, first consider the *forwarder* agent $\mathtt{Fw}\langle ab\rangle$

$$\mathtt{Fw}\langle ab\rangle \stackrel{\text{def}}{=} !a(x).\overline{b}\langle x\rangle$$

which repeatedly inputs a value at a and outputs it immediately at b. As another example, the following is a client which requests at a to have returned a value via a private name c

$$\overline{a}(c)\,c(y).P$$

where $\overline{a}(c)\,c(y).P$ stands for $(\nu c)(\overline{a}\langle c\rangle \,|\, c(y).P)$ with (νc) being a restriction operator. Using these agents, R below is a simple but interesting example of livelock

$$R \stackrel{\text{def}}{=} \mathtt{Fw}\langle aa\rangle \,|\, \overline{a}(c)\,c(y).P$$

since R causes an infinite reduction sequence and the receptor $c(y).P$ waits forever for an answer at c. In an untyped setting, R is equal to $\overline{a}(c)\,c(y).P$ up to asynchronous bisimilarity, but the two are quite different regarding resource consumption. The next example shows how subtleties arise through new link creation of the π-calculus.

$$a(x).\mathtt{Fw}\langle bx\rangle \,|\, \overline{a}(c)\mathtt{Fw}\langle cb\rangle \,|\, \overline{b}$$

After a one step reduction via a, we obtain $\mathtt{Fw}\langle bc\rangle \,|\, \mathtt{Fw}\langle cb\rangle \,|\, \overline{b}$ which exhibits divergence.

Type Discipline for SN The type discipline of this paper is a simple refinement of [8]. Concretely, the system is based on two central ideas:

- *Linear types* [12, 26, 27, 40], which ensure that a channel is used exactly once for input/output and, for a replicated channel, an input occurs exactly once and output occurs zero or more times [8, 24, 29, 32, 36].
- *Action types with causality*, where causality is represented by edges in a directed graph whose acyclicity ensures the absence of circular dependencies [26, 27, 40]. Transmission of causality is controlled by a form of *cut elimination* in action types.

Let us illustrate these points by examples. First, $\mathtt{Fw}\langle ab\rangle$ is typed as follows, assuming an appropriate environment Γ.

$$\Gamma \vdash \mathtt{Fw}\langle ab\rangle \triangleright\, !a \to ?b$$

Here $!a \to ?b$ indicates that the process repeatedly inputs at a and then outputs at b. Cut elimination occurs between input and output with the same name. For example, given an appropriate base Γ, we can type ($\otimes$ being disjoint union):

$$\Gamma \vdash \mathtt{Fw}\langle a_1 c\rangle \,|\, \mathtt{Fw}\langle a_2 c\rangle \,|\, \mathtt{Fw}\langle cb\rangle \triangleright\, !a_1 \to ?b \otimes !a_2 \to ?b \otimes !c \to ?b$$

$$\Gamma \vdash !a(x).(\overline{b}\langle x\rangle \,|\, \overline{b}\langle x\rangle) \,|\, !b(x).(\overline{c}\langle x\rangle \,|\, \overline{c}\langle x\rangle) \triangleright\, (!a \to ?c) \otimes (!b \to ?c)$$

We can detect a cyclic dependency such as $\mathtt{Fw}\langle ab\rangle \,|\, \mathtt{Fw}\langle ba\rangle$ by looking at their types $!a \to ?b$ and $!b \to ?a$ [20, 24, 40].

Proving SN for the π-Calculus To prove SN for typable processes, the first idea would be, in the light of the previous examples, to show that reduction steps follow a non-circular ordering on free channels, e.g. the reductions of $\overline{a}\langle v\rangle|\mathtt{Fw}\langle ab\rangle|\mathtt{Fw}\langle bc\rangle$ proceed at a, b and c in this order, but in $\overline{a}\langle v\rangle|\mathtt{Fw}\langle ab\rangle|\mathtt{Fw}\langle ba\rangle$ are repeated between a and b. However, due to creation of new links and replication of terms, both crucial features of π-calculi, such reasoning is infeasible, at least in its naive form. Consider the process

$$!a(x).(\overline{x}\langle v_1\rangle|\overline{x}\langle v_2\rangle) \,|\, \overline{a}(c)\mathtt{Fw}\langle cb\rangle \,|\, !b(x).(\overline{a}\langle x\rangle|\overline{a}\langle x\rangle) \quad (1)$$

which has type $!a \otimes !b$. The process owns reductions first at a, then at b, then at a again. Further, the number of redexes increases exponentially in its course, *but* the computation terminates. Such behaviour occurs when a process requests the same resource more than once in an interaction, e.g. in an encoding of the λ-term $\lambda xyz.((xz)(yz))$ [28]. The difficulty in analysing (1) can be seen by considering the following subterm of a one step descendant of (1).

$$(\nu c)(\overline{c}\langle v_1\rangle \,|\, \overline{c}\langle v_2\rangle \,|\, \mathtt{Fw}\langle cb\rangle)$$

It contains a chain $!c \to ?b$, which is difficult to determine before c is passed. In fact, if we naively represent causality

incorporating bound names in (1), there is a circular chain $a \to c \to b \to a$, although this cycle never arises in actual interaction. How can we then prove termination? Simple structural inductions would not be usable for the same reason they do not work in typed λ-calculi [6, 11].

The idea we use is suggested by SN proofs for typed λ-calculi, due to, among others, Tait [37]. His method employs a semantic interpretation of each type $[\![\sigma]\!]$ as a collection of strongly normalising λ-terms, and shows that all typable terms are indeed in these sets. A key step is to prove that $\lambda x:\sigma.M \in [\![\sigma \to \tau]\!]$ for each $M:\tau$ (for which by induction $M \in [\![\tau]\!]$), which means, by definition, $(\lambda x.M)N \in [\![\tau]\!]$ for each $N \in [\![\sigma]\!]$. But all semantic types have the property that $M\{N/x\} \in [\![\tau]\!]$ and $(\lambda x.M)N \longrightarrow M\{N/x\}$ imply $(\lambda x.M)N \in [\![\tau]\!]$, hence we have only to show $M\{N/x\} \in [\![\tau]\!]$. To be able to do this we strengthen the induction hypothesis $M \in [\![\tau]\!]$ to $M \in [\![\tau]\!]_\rho$ for each environment ρ, mapping each variable of type σ to some term in $[\![\sigma]\!]$. Now the result is immediate. While we cannot use an identical framework due to the different nature of reduction in the π-calculus, a similar technique works "for the induction to go through". A key observation concerns the close correspondence between the substitution $M\{N/x\}$ and the consumption of a message $\overline{x}\langle v\rangle$ by a replicated process $!x(y).Q$. Thus, at each induction step, we prove that $P|(R_1|...|R_n)$ converges for each possible "environment" $R_1|...|R_n$ which complements P. Termination behaviour is calculated via the extended reduction suggested by strong bisimilarity (which does not change termination) together with replication theorems [8, 30, 36]. Then acyclicity in causality yields strong normalisation.

Summary of Contributions The following summarises main technical contributions of the present work. (3) solves an open problem in [28] for the simple type hierarchy.

1. Introduction of a typed π-calculus where strong normalisability is ensured by typability. SN has significant consequences for the calculus, including the finite axiomatisation of the weak bisimilarity and the basic liveness in interaction.
2. Establishment of strong normalisability of typable processes combining ideas from traditional SN proofs for typed λ-calculi with process-theoretic reasoning.
3. Embedding, using Milner's encoding [28], of the simply typed λ-calculus with sums and products ($\lambda_{\to,\times,+}$) into our typed π-calculus. The embedding is fully abstract w.r.t. the observational congruence of $\lambda_{\to,\times,+}$, justifying all commutative conversions and η-equations [13] and automatically leads to SN in the source calculus.

Related Work Strong normalisation in typed λ-calculi has been studied extensively in the past; detailed surveys can be found in [6, 11]. Abramsky extends the Curry-Howard correspondence to linear logic [12] using proof expressions and proves SN [1], guiding our present usage of acyclicity in names. This programme is taken further with realisability semantics of linear logic in [5] where CCS processes act as realisers. The operational structure of [5] follows his own π-calculus encoding of proof nets [2]. The appeal of realisability lies in treating semantics and syntax uniformly on a logical basis. In the context of SN types for the π-calculus, sharing of names and dynamic link creation would make the framework in [1, 5] hard to apply directly. In contrast, the present work offers a possibly basic type discipline that does not directly correspond to known logical systems but is based on simple operational principles, resulting in a new effective method to ensure SN for name passing processes.

As our initial example of server-client interaction suggests, SN in processes is closely related to liveness properties in interaction. Yoshida [40] presents a typed π-calculus with a local liveness property. Kobayashi and his colleagues [23–26] propose several typing systems which ensure a form of liveness (in [25] time quotas are assigned to communications for this purpose). Unlike the present work, these and other preceding typing systems for π-calculus [8, 16, 17, 34, 36] do not guarantee SN and the associated liveness properties for processes involving non-trivial use of replication. As a result, embeddability of, say, $\lambda_{\to}$ in these systems does not guarantee the SN of the source calculus in these systems.

Structure of the Paper Section 2 introduces the syntax and the type discipline. Section 3 proves the main result, the strong normalisability. Section 4 presents the complete axiomatisation of weak bisimilarity. Section 5 gives a fully abstract encoding of $\lambda_{\to,\times,+}$. Section 6 briefly outlines further results. The technical details, including omitted proofs, can be found in the full version [41].

2. Processes and Typing

2.1. Processes

Following [8], we use the asynchronous version of the π-calculus [10, 19] with bound output [35]. [1]

P	$::=$	$x(\vec{y}).P$	input	$\mid$	$P \mid Q$	parallel
	$\mid$	$\overline{x}(\vec{y})P$	output	$\mid$	$(\nu x)P$	hiding
	$\mid$	$!x(\vec{y}).P$	replication	$\mid$	$\mathbf{0}$	inaction

The bound/free names are defined as usual and we assume the variable convention for bound names. The structural rules are standard except for omission of $!P \equiv !P|P$ and

[1] The full syntax includes branching, which is discussed in Section 5.2.

for incorporation of congruence rules making output asynchronous [8].

$$\begin{array}{lcll} \overline{x}(\vec{z})(P \mid Q) & \equiv & (\overline{x}(\vec{z})P) \mid Q & \text{if } \mathsf{fn}(Q) \cap \{\vec{z}\} = \emptyset \\ \overline{x}(\vec{z})(\nu w)P & \equiv & (\nu w)\overline{x}(\vec{z})P & \text{if } w \notin \{x\vec{z}\} \end{array}$$

The reduction $\longrightarrow$ is generated by the following rules.

$$\begin{array}{rcl} x(\vec{y}).P \mid \overline{x}(\vec{y})Q & \longrightarrow & (\nu \vec{y})(P \mid Q) \\ !x(\vec{y}).P \mid \overline{x}(\vec{y})Q & \longrightarrow & !x(\vec{y}).P \mid (\nu \vec{y})(P \mid Q) \end{array}$$

The relation is defined over processes modulo $\equiv$ and is closed under parallel composition, restriction and output.

2.2. Types

Channel Types The following pairs of *action modes* [8, 20] prescribe how each channel is used in typed processes.

$\downarrow$	Linear input	$\uparrow$	Linear output
!	Replicated input	?	Output to !

We also use $\perp$ to indicate the presence of both input and output at a linear channel. $p, q, \ldots$ range over action modes. For $p \neq \perp$, we write $\overline{p}$ for the *dual* of p, a self-inverse map on the action modes such that $\overline{\downarrow} = \uparrow$ and $\overline{!} = ?$. The modes correspond to $!_1, ?_1, !_\omega$ and $?_\omega$ introduced in [8], except that the present modes indicate true linearity for linear channels (i.e. input and output occurs precisely once) and lack of divergence for replicated channels.

Using action modes, we first define the set of *channel types*: they are assigned to names and indicate how channels would be used.

$$\begin{array}{lcllcll} \alpha & ::= & \langle \tau, \overline{\tau} \rangle & \tau_{\mathrm{I}} & ::= & (\vec{\tau}_{\mathrm{O}})^{\downarrow} & \mid \ (\vec{\tau}_{\mathrm{O}})^{!} \\ \tau & ::= & \tau_{\mathrm{I}} \mid \tau_{\mathrm{O}} & \tau_{\mathrm{O}} & ::= & (\vec{\tau}_{\mathrm{I}})^{\uparrow} & \mid \ (\vec{\tau}_{\mathrm{I}})^{?} \end{array}$$

In the first line $\overline{\tau}$ denotes the *dual* of τ, which is the result of dualising all action modes; $\mathsf{md}(\tau)$ indicates the (outermost) action mode of τ. A type of form $\langle \tau, \overline{\tau} \rangle$, called a *pair type*, is an unordered pair of mutually dual types.

Following [8] we only consider types where, in $(\vec{\tau}_{\mathrm{O}})^{\downarrow}$, each τ_i has mode ? (and dually for $(\vec{\tau}_{\mathrm{I}})^{\uparrow}$). This constraint, which comes from game semantics, is not essential for SN but simplifies presentation and proofs.

Action Types Channel types are assigned to free names of a process to specify possible usage of names. Action types, on the other hand, carry causality information [40] and witness the real usage of channels. Formally, an *action type*, denoted $A, B, \ldots$, is a finite directed graph with nodes of the form px, such that:

- no names occur twice; and
- edges are of the form $!x \to ?y$ or $\downarrow x \to \uparrow y$.

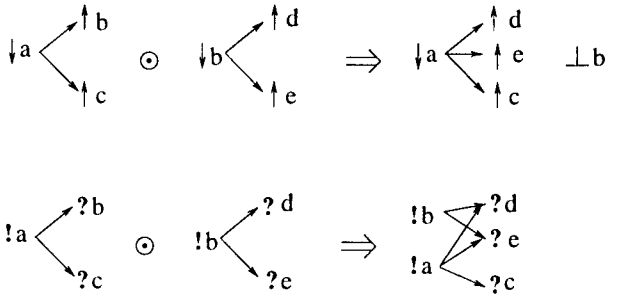


Figure 1. Composition of Action Types

If px is in A and for no y we have $qy \to px$, then we say x is *active in* A. $|A|$ (resp. $\mathsf{fn}(A)$, $\mathsf{active}(A)$, $\mathsf{md}(A)$) denotes the set of nodes (resp. names, active names, modes) in A. $A \backslash \vec{x}$ is the result of taking off nodes with names in $\vec{x}$ from A. $A \uplus B$ is the graph union of A and B.

Now define a symmetric partial operator $\odot$ by: $\downarrow \odot \uparrow = \perp$, $? \odot ? = ?$ and $! \odot ? = !$. Write $A \asymp B$ iff:

- whenever $px \in A$ and $qx \in B$, $p \odot q$ is defined; and
- whenever $p_1 x_1 \to \overline{p}_2 x_2, p_2 x_2 \to \overline{p}_3 x_3, \ldots, p_n x_n \to \overline{p}_{n+1} x_{n+1}$ in $A \uplus B$ $(n \geq 1)$, we have $x_1 \neq x_n$.

Then $A \odot B$, defined iff $A \asymp B$, is the following action type.

- $px \in |A \odot B|$ iff either (1) $px \in |A|$ and $x \notin \mathsf{fn}(B)$ and the symmetric case, or (2) $qx \in |A|$ and $rx \in |B|$ and $p = q \odot r$.
- $px \to qy$ in $A \odot B$ iff both (1) $px, qy \in |A \odot B|$ and (2) $px = r_1 z_1 \to \overline{r}_2 z_2, r_2 z_2 \to \overline{r}_3 z_3, \ldots, r_n z_n \to \overline{r}_{n+1} z_{n+1} = qy$ in $A \uplus B$ $(n \geq 1)$.

We can easily check that $\odot$ is a symmetric and associative partial operation on action types with unit $\emptyset$.

Example 2.1 Figure 1 shows examples of composition between action types. In the linear case, ordering from/to node b disappears. On the other hand, in the replicated input case, we need to keep the original ordering because $!b(\vec{y}).P$ remains persistently. We can write down these examples syntactically as follows (shared ?-nodes are duplicated in syntax): $\downarrow a \to (\uparrow b \otimes \uparrow c) \odot \downarrow b \to (\uparrow d \otimes \uparrow e) = \downarrow a \to (\uparrow c \otimes \uparrow d \otimes \uparrow e) \otimes \perp b$, and $!a \to (?b \otimes ?c) \odot !b \to (?d \otimes ?e) = !a \to (?c \otimes ?d \otimes ?e) \otimes !b \to (?d \otimes ?e)$.

2.3. Linear Typing

We are now ready to present the typing rules. Sequents have the form $\Gamma \vdash P \triangleright A$ where Γ is a finite map from names to

$$\frac{}{\Gamma \vdash \mathbf{0} \triangleright \emptyset}\ (\text{Zero}) \qquad \frac{\Gamma \vdash P_i \triangleright A_i \quad (i=1,2) \quad A_1 \asymp A_2}{\Gamma \vdash P_1 \mid P_2 \triangleright A_1 \odot A_2}\ (\text{Par}) \qquad \frac{\Gamma \cdot x:\alpha \vdash P \triangleright A \quad px \in |A| \text{ with } p \in \{\bot,!\}}{\Gamma \vdash (\nu x{:}\alpha)P \triangleright A/x}\ (\text{Res})$$

$$\frac{\Gamma \vdash x:\downarrow,\uparrow \quad \Gamma \vdash P \triangleright A^{-x}}{\Gamma \vdash P \triangleright A \otimes \bot x}\ (\text{Weak-}\bot) \qquad \frac{\Gamma \vdash x:? \quad \Gamma \vdash P \triangleright A^{-x}}{\Gamma \vdash P \triangleright A \otimes ?x}\ (\text{Weak-}?)$$

$$\frac{\Gamma \vdash x:(\vec{\tau})^{\downarrow} \quad \Gamma \cdot \vec{y}:\vec{\tau} \vdash P \triangleright C^{-x}}{\Gamma \vdash x(\vec{y}:\vec{\tau}).P \triangleright (\downarrow x \to A) \otimes B}\ (\text{In}^{\downarrow})\ (C/\vec{y} = \uparrow A \otimes ?B) \qquad \frac{\Gamma \vdash x:(\vec{\tau})^{\uparrow} \quad \Gamma \cdot \vec{y}:\vec{\tau} \vdash P \triangleright C}{\Gamma \vdash \overline{x}(\vec{y}:\vec{\tau})P \triangleright A \odot \uparrow x}\ (\text{Out}^{\uparrow})\ (C/\vec{y} = A \asymp \uparrow x)$$

$$\frac{\Gamma \vdash x:(\vec{\tau})^{!} \quad \Gamma \cdot \vec{y}:\vec{\tau} \vdash P \triangleright C^{-x}}{\Gamma \vdash !x(\vec{y}:\vec{\tau}).P \triangleright !x \to A}\ (\text{In}^{!})\ (C/\vec{y} = ?A) \qquad \frac{\Gamma \vdash x:(\vec{\tau})^{?} \quad \Gamma \cdot \vec{y}:\vec{\tau} \vdash P \triangleright C}{\Gamma \vdash \overline{x}(\vec{y}:\vec{\tau})P \triangleright A \odot ?x}\ (\text{Out}^{?})\ (C/\vec{y} = A \asymp ?x)$$

Figure 2. Linear Typing Rules

channel types, called a *base*. The typing rules are given in Figure 2. The following notation is used.

$A/\vec{x}$	$A\backslash\vec{x}$ such that $x_i \in \mathsf{active}(A)$ for each x_i
pA	A such that $\mathsf{md}(A) = \{p\}$
A^{-x}	A such that $x \notin \mathsf{fn}(A)$

Further, $px \to A$ adds new edges from px to active nodes in A, $A \otimes B$ (resp. $\Gamma \cdot \Delta$) denotes the disjoint union of A and B (resp. Γ and Δ), and $\Gamma \vdash x:\tau$ denotes either $x:\tau$ or $x:\langle\tau,\overline{\tau}\rangle$ is in Γ. The sequent $\Gamma \vdash P \triangleright A$ is often abbreviated to $\Gamma \vdash P$.

We briefly illustrate the typing rules. In (Par), "$\asymp$" controls composability, ensuring linearity of channels, and prevents circular dependency. In (Res), we do not allow $\uparrow$, ? or $\downarrow$-channels to be restricted since they expect their dual actions to exist in the environment (cf. [8, 17, 20, 26]). In addition to recording causality, ($\text{In}^{\downarrow}$) ensures that x occurs precisely once (by C^{-x}) and that no free input is suppressed under the prefix. ($\text{Out}^{\uparrow}$) also ensures that x occurs precisely once but permits suppression by the prefix since output is asynchronous. ($\text{In}^{!}$) is the same as ($\text{In}^{\downarrow}$) except that no free $\uparrow$-channels are suppressed (if a $\uparrow$-channel is under replication then it can be used more than once). ($\text{Out}^{?}$) and (Weak-?) say that ?-channels occur zero or more times and do not suppress actions.

Example 2.2

- A *copy-cat* copies all information from one channel to another [4, 22]. We show, step by step, how $[u \to x]^{\tau} \stackrel{\text{def}}{=} !u(a).\overline{x}(b)b.\overline{a}$, the copy-cat from u to x, can be typed. Let $\tau = (()^{\uparrow})^{!}$, $\Gamma = a:()^{\uparrow} \cdot b:()^{\downarrow} \cdot u:\tau \cdot x:\overline{\tau}$. Then: (1) $\Gamma \vdash \mathbf{0} \triangleright \emptyset$, (2) $\Gamma \vdash \overline{a} \triangleright \uparrow a$, (3) $\Gamma \vdash b.\overline{a} \triangleright \downarrow b \to \uparrow a$, (4) $\Gamma\backslash b \vdash \overline{x}(b)b.\overline{a} \triangleright ?x \otimes \uparrow a$ (by $(\downarrow b \to \uparrow a)/b = \uparrow a$) and finally (5) $\Gamma\backslash ab \vdash !u(a).\overline{x}(b)b.\overline{a} \triangleright !u \to ?x$ (by $(?x \otimes \uparrow a)/a = ?x$).
- Let $\alpha = \langle()^{\downarrow}, ()^{\uparrow}\rangle$ and $\Gamma = a:\alpha \cdot b:\alpha \cdot c:\alpha \cdot d:\alpha$. Then $\Gamma \vdash a.(\overline{b} \mid \overline{c}) \triangleright \downarrow a \to (\uparrow b \otimes \uparrow c)$ and $\Gamma \vdash b.\overline{d} \triangleright \downarrow b \to \uparrow d$. By (Par), $\Gamma \vdash a.(\overline{b} \mid \overline{c}) \mid b.\overline{d} \triangleright \downarrow a \to (\uparrow c \otimes \uparrow d) \otimes \bot b$, and by (Res), $\Gamma \vdash (\nu b)(a.(\overline{b} \mid \overline{c}) \mid b.\overline{d}) \triangleright \downarrow a \to (\uparrow c \otimes \uparrow d)$.
- Let $\Gamma = x:\langle\tau,\overline{\tau}\rangle \cdot y:\langle\tau,\overline{\tau}\rangle \cdot z:\langle\tau,\overline{\tau}\rangle$ and $\tau = (()^{\uparrow})^{!}$. Then the connection of two links (copy-cats) is typed as:

$$\Gamma \vdash [x \to y]^{\tau} \mid [y \to z]^{\tau} \triangleright (!x \to ?z) \otimes (!y \to ?z)$$

with $(!x \to ?y) \odot (!y \to ?z) = (!x \to ?z) \otimes (!y \to ?z)$. However, $[x \to x]^{\tau}$ and $[x \to y]^{\tau} \mid [y \to x]^{\tau}$ are untypable under any environment by the side condition C^{-x} in ($\text{In}^{!}$) and by definition of $\asymp$, respectively.

Next we list two properties of name usage in typed processes. Acyclicity becomes crucial in our SN proof later.

Proposition 2.3 *Let $\Gamma \vdash P \triangleright A$.*

i. (linearity) *If $px \in A$ such that $p \in \{\downarrow,\uparrow,!\}$, then x occurs precisely once in P.*

ii. (acyclicity) *$G(P)$ denotes a directed graph s.t.: (1) nodes are $\mathsf{fn}(P)$; and (2) edges are given by: $x \curvearrowright y$ iff $P \equiv (\nu\vec{z})(Q|R)$ such that $Q \equiv x(\vec{w}).Q_0$ or $Q \equiv !x(\vec{w}).Q_0$ where y occurs free in Q_0, $x \notin \{\vec{z}\}$ and $y \notin \{\vec{z}\vec{w}\}$. A* cycle *in $G(P)$ is a sequence of form $x \curvearrowright y_1 \ldots \curvearrowright y_n \curvearrowright x$ ($n \geq 0$) with $y_i \neq x$. Then $G(P)$ has no cycle.*

Some notation which we use later:

- $P \Downarrow Q \stackrel{\text{def}}{\Leftrightarrow} P \longrightarrow^{*} Q \not\longrightarrow$.
- $P \downarrow \stackrel{\text{def}}{\Leftrightarrow} \exists Q.P \Downarrow Q$. Further, $P \Uparrow \stackrel{\text{def}}{\Leftrightarrow} \forall n \in \mathbb{N}.\ P \longrightarrow^{n}$.
- $\mathsf{SN}(P) \stackrel{\text{def}}{\Leftrightarrow} \neg P \Uparrow$.
- $\mathsf{CSN}(P) \stackrel{\text{def}}{\Leftrightarrow} \mathsf{SN}(P) \wedge (P \Downarrow Q_{1,2} \Rightarrow Q_1 \equiv Q_2)$.

Proposition 2.4 *Let $\Gamma \vdash P \triangleright A$.*

i. (subject reduction) *If $P \longrightarrow^{*} Q$ then $\Gamma \vdash Q \triangleright A$.*

ii. (one-step confluence) *If $P \longrightarrow Q_i$ ($i = 1,2$) with $Q_1 \not\equiv Q_2$ then there exists R s.t. $Q_i \longrightarrow R$ ($i = 1,2$).*

iii. (determinacy) (1) $P \longrightarrow P'$ *and* $\mathsf{SN}(P')$ *imply* $\mathsf{SN}(P)$. (2) $P \Downarrow Q_i$ $(i = 1,2)$ *imply* $Q_1 \equiv Q_2$. *And* (3) $P \Downarrow \Leftrightarrow \mathsf{SN}(P) \Leftrightarrow \mathsf{CSN}(P)$.

(i,ii) is proved as in [8]. (iii) is standard [1] all using (ii).

3. Strong Normalisation

This section proves the following result.

Theorem 3.1 *(main theorem, strong normalisation)* $\Gamma \vdash P \triangleright A \;\Rightarrow\; \mathsf{CSN}(P)$

A few significant consequences of the theorem will be discussed in Sections 4, 5 and 6. In the proof, we first introduce the *extended reduction relation* $\mapsto$, which eliminates all *cuts* (mutually dual channels) in a typed process. Next we define *semantic types* $[\![\Gamma, A]\!]$, which are sets of typed terms that converge when composed with all necessary "resources" (i.e. complementary processes). Finally we prove that each typable process is in the corresponding semantic type. This part is divided into two stages. We start with show all normal forms to be in their semantic types. Then we establish that each typable process combined with resources always reaches a normal form, which implies the strong normalisability of $\longrightarrow$. In the second stage acyclicity (cf. Proposition 2.3) becomes crucial.

3.1. Extended Reductions

Definition 3.2 (extended reductions) We define $\mapsto_{\mathsf{l}}$, $\mapsto_{\mathsf{r}}$ and $\mapsto_{\mathsf{g}}$ as the compatible relations on processes modulo $\equiv$ respectively generated by the following rules.

$$\begin{array}{rcl} C[\overline{x}(\vec{y})P] | x(\vec{y}).Q & \mapsto_{\mathsf{l}} & C[(\nu \vec{y})(P|Q)] \\ C[\overline{x}(\vec{y})P] | !x(\vec{y}).Q & \mapsto_{\mathsf{r}} & C[(\nu \vec{y})(P|Q)] \,|\, !x(\vec{y}).Q \\ (\nu x)!x(\vec{y}).Q & \mapsto_{\mathsf{g}} & \mathbf{0} \end{array}$$

Here $C[\,]$ is an arbitrary context not binding x. Then $\mapsto \stackrel{\text{def}}{=} (\mapsto_{\mathsf{l}} \cup \mapsto_{\mathsf{r}} \cup \mapsto_{\mathsf{g}})$ is the *extended reduction* relation.

The idea of $\mapsto$ is to capture known process-algebraic laws as one step reductions: $\mapsto_{\mathsf{l}}$, $\mapsto_{\mathsf{r}}$ and $\mapsto_{\mathsf{g}}$ correspond to the β/linear law [16, 17, 26, 40], the replication law [8, 32, 36] and the garbage collection law, respectively. Immediately $\longrightarrow \subseteq \mapsto^*$. $P \Downarrow_e$, $\mathsf{SN}_e(P)$ and $\mathsf{CSN}_e(P)$ are given as $P \Downarrow$, $\mathsf{SN}(P)$ and $\mathsf{CSN}(P)$, using $\mapsto$ instead of $\longrightarrow$. A $\mapsto$-*redex* is a pair of terms which form a redex for $\mapsto$ in a given term. We say process P is *prime with subject* x if either P is input with subject at x or $P \equiv \overline{x}(y_1..y_n)\Pi_{i\in I}P_i$ such that each P_i is prime with subject y_i where $\Pi_{i\in I}P_i$ denotes the parallel composition of $\{P_i\}_{i\in I}$ (if $I = \emptyset$ then $\Pi_{i\in I}P_i = \mathbf{0}$). We assume all prefixed terms to be primes throughout the rest of the section (which does not lose generality up to $\equiv$). NF_e is given by $\{\Gamma \vdash P, P \not\mapsto\}$. Note that a process is in NF_e if it does not contain complementary input and output and, moreover, it does not have substantial hiding (i.e. a hiding $(\nu x)P$ such that $x \in \mathsf{fn}(P)$). Thus we can see NF_e is inductively generated by the following rules up to $\equiv$ (implicitly assuming typability):

- $\mathbf{0} \in \mathsf{NF}_e$,
- $P \in \mathsf{NF}_e$ then $x(\vec{y}:\vec{\tau}).P$, $!x(\vec{y}:\vec{\tau}).P$, $\overline{x}(\vec{y}:\vec{\tau})P \in \mathsf{NF}_e$.
- $P_i \in \mathsf{NF}_e$ $(i \in I \neq \emptyset)$, P_i is a prime, and $P_i | P_j \not\mapsto$ $(i \neq j)$ then $\Pi_{i\in I}P_i \in \mathsf{NF}_e$.

Proposition 3.3 *Let all processes be typed below.*

i. *If* $\Gamma \vdash P \triangleright A$ *and* $P \mapsto P'$ *then* $\Gamma \vdash P' \triangleright A$.

ii. (CR) *If* $P \mapsto^* Q_i$ *then* $Q_i \mapsto^* R$ $(i = 1,2)$.

iii. (determinacy) *If* $P \mapsto P'$ *and* $\mathsf{SN}_e(P')$ *then* $\mathsf{SN}_e(P)$. *Thus* $P \Downarrow_e$ *iff* $\mathsf{SN}_e(P)$ *iff* $\mathsf{CSN}_e(P)$.

Note that the Church-Rosser property is no longer one-step. The proof proceeds by 'postponing' applications of $\mapsto_{\mathsf{g}}$.

3.2. Semantic Types

Semantic types are provably strongly normalising typed terms of some kind. We need some preliminaries.

- $\mathsf{c}(A) \stackrel{\text{def}}{=} \otimes_{p_i x_i : A, p_i \in \{\uparrow,?\}} \overline{p_i} x_i$.
- Let $A \asymp B$ and $A \odot B = C \otimes \perp\vec{x}$ where $\perp \notin \mathsf{md}(C)$. Then $A \cdot B \stackrel{\text{def}}{=} C$.

By $\mathsf{c}(A)$, called the *complement of* A, we indicate the (type of the) environment which gives complementary linear and replicated inputs for all free output channels in A. $A \cdot B$ is a "semantic version" of $A \odot B$, where we forget inessential $\perp$-channels. We write $\Gamma \vdash A$ if modes in A conform to Γ. We can now define semantic types.

Definition 3.4 The *semantic type* $[\![\Gamma, A]\!]$ of a pair Γ and A such that $\Gamma \vdash A$, and the *prime semantic type* $\langle\langle \Gamma, px \rangle\rangle$ for a pair Γ and px such that $\Gamma \vdash px$ with $p \in \{\downarrow, !\}$, are defined by the rules in Figure 3.

In Figure 3, $\overline{\Gamma}$ and $\Gamma \cup \overline{\Gamma}$ respectively denote the result of dualising all types in Γ and the name-wise union of types. The rules are well-defined since the height of types decreases in effect. Note that we can always assume Γ in $[\![\Gamma, A]\!]$ is *paired*, i.e. contains only pair types, with no loss of generality. Some observations:

Lemma 3.5

i. *If* $P \in [\![\Gamma, A]\!]$ *then* $\Gamma \vdash P \triangleright A$ *and* $\mathsf{SN}_e(P)$.

ii. $[\![\Gamma, A]\!] \subseteq [\![\Gamma, A \otimes B]\!]$. *Also* $[\![\Gamma, A \otimes \perp x]\!] = [\![\Gamma, A]\!]$.

$$
\begin{array}{rcl}
[\![\Gamma, A]\!] & \stackrel{\text{def}}{=} & \{\Gamma \vdash P \triangleright A \mid \forall Q \in \langle\langle \overline{\Gamma}, \mathsf{c}(A)\rangle\rangle.\, P|Q \Downarrow_e R \in \langle\langle \Gamma \cup \overline{\Gamma}, A \cdot \mathsf{c}(A)\rangle\rangle\} \\
\langle\langle \Gamma, \downarrow x\rangle\rangle & \stackrel{\text{def}}{=} & \{x(\vec{y}:\vec{\tau}).P \mid P \in [\![\Gamma \cdot \vec{y}:\vec{\tau}, \otimes p_i y_i]\!] \text{ with } \Gamma \vdash x : (\vec{\tau})^{\downarrow} \text{ and } p_i = \mathsf{md}(\tau_i)\} \\
\langle\langle \Gamma, !x\rangle\rangle & \stackrel{\text{def}}{=} & \{!x(\vec{y}:\vec{\tau}).P \mid P \in [\![\Gamma \cdot \vec{y}:\vec{\tau}, \otimes p_i y_i]\!] \text{ with } \Gamma \vdash x : (\vec{\tau})^{!} \text{ and } p_i = \mathsf{md}(\tau_i)\} \\
\langle\langle \Gamma, \otimes_{i\in I} p_i x_i\rangle\rangle & \stackrel{\text{def}}{=} & \{\Pi_{i\in I} P_i \mid P_i \in \langle\langle \Gamma, p_i x_i\rangle\rangle \quad (i \in I)\}
\end{array}
$$

Figure 3. None-Prime and Prime Semantic Types

iii. *Let* $P \mapsto P'$. *Then* $P \in [\![\Gamma, A]\!]$ *iff* $P' \in [\![\Gamma, A]\!]$.

iv. *Let* $P_i \in \langle\langle \Gamma, p_i x_i\rangle\rangle$ $(1 \le i \le n)$ *such that* $x_1, .., x_n$ *are pairwise distinct. Then* $\Pi_{i\in I} P_i \in [\![\Gamma, \otimes_i p_i x_i]\!]$.

For the proof of (i), we use $P|Q \Downarrow_e$ implies $P \Downarrow_e$ and $Q \Downarrow_e$. For (iii), "then" is trivial, while "if" is by $\mapsto$ being CR. (iv) is because $\mathsf{c}(\otimes p_i x_i) = \emptyset$ in this case.

3.3. Main Proofs

First we show that all (typable) normal forms are semantically typed. The difficult case here is output $\overline{a}(\vec{x})P$ to replication $!a(\vec{x}).Q$ because after reduction $\overline{a}(\vec{x})P \mid !a(\vec{x}).Q \longrightarrow (\nu\vec{x})(P|Q) \mid !a(\vec{x}).Q$, P may interact again with $!a(\vec{x}).Q$. Our formulation of semantic types based on $\mapsto$ makes the inductive augument possible.

Lemma 3.6 *If* $\Gamma \vdash P \triangleright A$ *and* $P \in \mathsf{NF}_e$ *then* $P \in [\![\Gamma, A]\!]$.

PROOF: By Lemma 3.5 (ii), it suffices to consider only minimum action types. For brevity we write $P_{\langle px\rangle}$ $(p \in \{!, \downarrow\})$ for a process in normal form in a prime semantic type. Also throughout the proof we set $\mathsf{fn}(A) = \{a_i\}$ and $\mathsf{fn}(B) = \{b_j\}$. The proof proceeds by induction on the structure of P. We only list two cases, see [41] for the remaining cases.

(Inaction). By $\mathsf{c}(\emptyset) = \emptyset$, if $Q \in \langle\langle \Gamma, \mathsf{c}(\emptyset)\rangle\rangle$, then $Q \equiv \mathbf{0}$. Hence $\mathbf{0}|Q \equiv \mathbf{0} \Downarrow_e \mathbf{0} \in \langle\langle \Gamma, \emptyset\rangle\rangle$ with $\mathsf{c}(\emptyset) \cdot \emptyset = \emptyset$, immediately $\mathbf{0} \in [\![\Gamma, \emptyset]\!]$.

(Replicated Output). Assume $P \in [\![\Gamma \cdot \vec{y} : \vec{\tau}, C \otimes ?x]\!]$ with $C/\vec{y} = \uparrow A \otimes ?B^{-x}$. We have to show $\overline{x}(\vec{y} : \vec{\tau})P \in [\![\Gamma, A \otimes B \otimes ?x]\!]$. First we note that $\mathsf{c}(A \otimes B \otimes ?x) = \mathsf{c}(C \otimes ?x) = (\overline{A} \otimes \overline{B} \otimes !x)$. Assume $Q \in \langle\langle \Gamma, \overline{A} \otimes \overline{B} \otimes !x\rangle\rangle$. W.l.o.g. we can write $Q \equiv !x(\vec{y}).Q_0' \mid Q_1 \mid Q_2$ where $!x(\vec{y}).Q_0' \in \langle\langle \Gamma, !x\rangle\rangle$, $Q_1 \equiv \Pi_i Q_{1i\langle\downarrow a_i\rangle}$ and $Q_2 \equiv \Pi_j Q_{2j\langle !b_j\rangle}$. Then we have:

$$\overline{x}(\vec{y})P \mid Q \longrightarrow (\nu\vec{y})(P|Q_0') \mid !x(\vec{y}).Q_0' \mid Q_1 \mid Q_2.$$

By induction, $P|!x(\vec{y}).Q_0'|Q_1|Q_2 \Downarrow_e P'|!x(\vec{y}).Q_0'|Q_2$ s.t. $P' \in [\![\Gamma \cdot \vec{y} : \vec{\tau}, \otimes p_i y_i]\!]$ with $p_i = \mathsf{md}(\tau_i) \in \{!, \downarrow\}$. Hence we can write $P' \equiv \Pi_k R_{1k\langle\downarrow z_k\rangle} \mid \Pi_l R_{2l\langle !w_l\rangle}$ with $\{\vec{y}\} = \{\vec{z}\vec{w}\}$. We also note that $Q_0' \in \langle\langle \Gamma \cdot \vec{y} : \vec{\tau}, \otimes \overline{p}_i y_i\rangle\rangle$. Hence, by assumption,

$$(\nu\vec{y})(P'|Q_0') \mapsto^* (\nu\vec{y})(\Pi_l R_{2l\langle !w_l\rangle}) \mapsto_{\mathsf{g}}^* \mathbf{0}$$

Now by CR, we have $P \mid Q \Downarrow_e !x(\vec{y}).Q_0' \mid Q_2 \in \langle\langle \Gamma, \overline{B} \otimes !x\rangle\rangle$, as desired. ■

Corollary 3.7 *If* $\Gamma \vdash P \triangleright px \in \mathsf{NF}_e$ *with* $p \in \{\downarrow, !\}$, *then* $P \in \langle\langle \Gamma, px\rangle\rangle$.

We can now establish the main lemma below: given the Lemma 3.6, prefix and restriction become trivial, but parallel composition causes problems. Even if $!a.\overline{b}$ and $(\overline{a} \mid !b.\overline{c})$ are in NF_e, their composition (with environment $!c.\mathbf{0}$) allows reductions. How can we prove termination? The key idea is to contract $\mapsto$-redexes from the end of the order of names $c \curvearrowleft b \curvearrowleft a$ as:

$$
\begin{array}{rcccl}
 & & !a.\overline{b} \mid \overline{a} \mid !b.\overline{c} \mid !c.\mathbf{0} & \mapsto_{\mathsf{r}} & !a.\overline{b} \mid \overline{a} \mid !b.\mathbf{0} \mid !c.\mathbf{0} \\
\mapsto_{\mathsf{r}} & & !a.\mathbf{0} \mid \overline{a} \mid !b.\mathbf{0} \mid !c.\mathbf{0} & \mapsto_{\mathsf{r}} & !a.\mathbf{0} \mid !b.\mathbf{0} \mid !c.\mathbf{0}
\end{array}
$$

This reduction strategy always terminates due to acyclicity of names. Formally, we prove:

Lemma 3.8 (main lemma) *Suppose* $\Gamma \vdash P \triangleright A$. *Then* $P|Q \Downarrow_e$ *for each* $Q \in [\![\overline{\Gamma}, \mathsf{c}(A)]\!]$.

PROOF: By induction on the typing rules. (Zero) and (Weak-$\perp$,-?) are trivial. For the prefix rules, by induction the body of each prefix converges, hence so does the whole term. Then we use Lemma 3.6 again. (Res) is similar, by Lemma 3.5 (ii). For (Par), suppose $\Gamma \vdash P_i \triangleright A_i$ with $i = 1, 2$ such that $A_1 \asymp A_2$ and let $A = A_1 \odot A_2$. By induction hypothesis $P_1 \Downarrow_e P_1'$ and $P_2 \Downarrow_e P_2'$. Let $P \stackrel{\text{def}}{=} P_1'|P_2'$. Then $P \equiv Q_1|..|Q_n$ where each Q_i is prime. If $n = 0$ there is nothing to prove. Assume $n \gtrsim 0$ and let $X \stackrel{\text{def}}{=} \{1, 2, .., n\}$. We define the relation $\searrow$ on X as follows:

$$i \searrow j \stackrel{\text{def}}{\Leftrightarrow} \exists x \in \mathsf{fn}(Q_i), y \in \mathsf{fn}(Q_j).\, x \frown y$$

Since $i \searrow^+ j \searrow^+ i$ implies the existence of a cycle $x \frown^+ x$ in the sense of Proposition 2.3 (ii), $\searrow^*$ is a partial order on X. We now define a series of sets $X_1, X_2, ..$ as follows, writing $\mathsf{max}(Y, \le)$ for the set of maximal elements of a partially ordered set Y.

$$X_1 \stackrel{\text{def}}{=} \mathsf{max}(X, \searrow^*) \qquad X_{i+1} \stackrel{\text{def}}{=} \mathsf{max}(X \backslash \bigcup_{1 \le j \le i} X_i, \searrow^*)$$

As X is finite, $X_1,..,X_m$ partition X for some m. Now let $S_i \stackrel{\text{def}}{=} \Pi_{j\in X_i} Q_j$ for $1 \le i \le m$. Then $P \equiv \Pi_{1\le i\le m} S_i$ and $S_i \in \mathsf{NF}_e$ for each i. Note the series $S_1,..,S_n$ is constructed so that outputs in S_{i+1} are always complemented by inputs in $S_i|S_{i-1}|...|S_1|R$. Now let $\Gamma \vdash S_i \triangleright C_i$ s.t. $\odot_{1\le i\le m} C_i = A$ and let $E_i \stackrel{\text{def}}{=} \mathsf{c}(C_1)\odot C_1 \odot .. \odot C_{i-1}$ for $1 \le i \le m$. Then $E_i = \mathsf{c}(C_i)$ for each i. Note also $E_1 = \mathsf{c}(A)$ and $E_m = \mathsf{c}(A)\odot A$. Choose any $R \in \langle\langle \Gamma,\ \mathsf{c}(A)\rangle\rangle$. We now show, by induction on $1 \le l \le m+1$, that for some $R_l \in \langle\langle \Gamma,\ E_l\rangle\rangle$

$$P|R \mapsto^* \Pi_{l\le i\le m} S_i | R_l.$$

This proves the lemma when $l = m+1$. For the base case, take $R_1 \equiv R$. For the inductive step, assume $P|R \mapsto^* \Pi_{l\le i\le m} S_i|R_l$ such that $R_l \in \langle\langle \Gamma,\ E_l\rangle\rangle$. By Lemma 3.6 and by $S_l \in \mathsf{NF}_e$ we know that $S_l \in [\![\Gamma,\ C_l]\!]$. By $E_l = \mathsf{c}(C_l) = \mathsf{c}(C_1)\odot C_1 \odot ... \odot C_{l-1}$, this implies $S_l|R_l \Downarrow_e R' \in \langle\langle \Gamma,\ E_{l+1}\rangle\rangle$. We can now set $R' \equiv R_{l+1}$, as desired. ■

Theorem 3.9 (strong normalisability in $\mapsto$) $\Gamma \vdash P \triangleright A$ *implies* $\mathsf{CSN}_e(P)$.

By $\longrightarrow \subsetneq \mapsto$ and Proposition 2.4 (iii-3), we have now established Theorem 3.1.

4. Characterisation of Bisimilarity

As a striking consequence of the strong normalisability of typed processes, this section shows that weak bisimilarity has a finite axiomatisation.

4.1. Typed Transitions and Typed Bisimulations

Typed transitions describe the observations a typed observer can make of a typed process. Typed transitions are a proper subset of untyped transitions while not restricting τ-actions: hence typed transitions restricts observability, not computation. First, *untyped transitions* $P \stackrel{l}{\longrightarrow} Q$, with labels τ, $x(\vec{y})$ and $\overline{x}(\vec{y})$ are generated by the following rules.

$$x(\vec{y}).P \stackrel{x(\vec{y})}{\longrightarrow} P \quad \overline{x}(\vec{z})P \stackrel{\overline{x}(\vec{z})}{\longrightarrow} P \quad !x(\vec{y}).P \stackrel{x(\vec{y})}{\longrightarrow} P|!x(\vec{y}).P$$

The communication and contextual rules are standard except for closure under asynchronous output.

$$P \stackrel{l}{\longrightarrow} P' \text{ with } \mathsf{fn}(l)\cap\{\vec{y}\} = \emptyset \ \Rightarrow\ \overline{x}(\vec{y})P \stackrel{l}{\longrightarrow} \overline{x}(\vec{y})P'$$

Typed transitions, written $\Gamma \vdash P \stackrel{l}{\longrightarrow} \Gamma\cdot\vec{y}:\vec{\tau} \vdash Q$, where $\vec{y}:\vec{\tau}$ assigns names introduced in l as prescribed by Γ, are generated as follows, cf. Section 4.2 and Appendix E of [8]: $\Gamma \vdash P \stackrel{l}{\longrightarrow} \Gamma\cdot\vec{y}:\vec{\tau} \vdash Q$ iff (1) $\Gamma \vdash P \triangleright A$, (2) $P \stackrel{l}{\longrightarrow} Q$ with $\mathsf{bn}(l)\cap\mathsf{fn}(\Gamma) = \emptyset$, (3) if $\perp x \in |A|$ then $\mathsf{fn}(l) \ne x$, and (4) if $!x \in |A|$ and $\mathsf{active}(l) = x$ then l is input.

Using typed transitions, we define bisimulations. Let us say a relation over typed processes is *typed* if it relates only processes with identical base and action type. A typed relation is a *typed congruence* when it is a typed equivalence closed under typed contexts, contains $\equiv$ and allows weakening of bases in the standard way [8, 32]. A typed relation $\mathbf{R}$ is a *weak bisimulation*, or a *bisimulation*, if $\Gamma \vdash P\mathbf{R}Q$ implies: whenever $\Gamma \vdash P \stackrel{l}{\longrightarrow} P'$ then there is a typed transition sequence $\Gamma \vdash Q \stackrel{\hat{l}}{\Longrightarrow} Q'$ such that $P\mathbf{R}Q$, as well as the symmetric case. By replacing $\stackrel{\hat{l}}{\Longrightarrow}$ with $\stackrel{l}{\longrightarrow}$, we obtain a *strong bisimulation*. If $\Gamma \vdash P\mathbf{R}Q$ for some weak (resp. strong) bisimulation $\mathbf{R}$, we write $\Gamma \vdash P \approx Q$ (resp. $\Gamma \vdash P \sim Q$). Finally, $\approx$ (resp. $\sim$) is called *weak* (resp. *strong*) *bisimilarity*. The weak bisimilarity is often simply called *bisimilarity*.

4.2. Characterisation

Let $\leftrightarrow$ be the transitive, symmetric closure of $\mapsto \cup \equiv$. We now show that $\leftrightarrow$ completely characterises bisimilarity.

Definition 4.1

- The relation $\equiv'$ is the least congruence such that $\equiv_\alpha \subseteq \equiv'$, $P|Q \equiv' Q|P$ and $(P|Q)|R \equiv' P|(Q|R)$.
- The relation $\triangleright$ is the least typed precongruence containing $\equiv'$ such that $P|\mathbf{0} \triangleright P$, $(\nu x)\mathbf{0} \triangleright \mathbf{0}$, $(\nu x)(P|Q) \triangleright P|(\nu x)Q$ if $x \notin \mathsf{fn}(P)$, $\overline{x}(\vec{y})(P|Q) \triangleright P|\overline{x}(\vec{y})Q$ if $\mathsf{fn}(P)\cap\{\vec{y}\} = \emptyset$ and $(\nu z)\overline{x}(\vec{y})P \triangleright \overline{x}(\vec{y})(\nu z)P$ if $z \notin \{x,\vec{y}\}$.
- P is in $\triangleright$-*normal form* if $P \in \mathsf{NF}_e$ and $P \triangleright Q$ implies $P \equiv' Q$.

Clearly $\triangleright$-normal forms are representatives of NF_e, in fact precisely those generated by the rules in §3.1.

Lemma 4.2 i. *If* $\Gamma \vdash P \triangleright A$ *then there is a* $\triangleright$*-normal form* Q *such that* $P \mapsto^* Q$.

ii. *Let* P *and* Q *be two typable* $\triangleright$*-normal forms. Then* $P \approx Q$ *iff* $P \equiv' Q$ *iff* $P \leftrightarrow Q$.

The proof of Lemma 4.2 uses Theorem 3.9. The key observation for the proof of (ii) is that $\triangleright$-normal forms are a class of processes where trace equivalence, $\approx$ and $\equiv'$ (hence also $\sim$ and $\equiv$) coincide.

Theorem 4.3 *(characterisation of* $\approx$*)* (i) $\mapsto \subsetneq \approx$, $\mapsto^* \subsetneq \approx$; *and* (ii) $\leftrightarrow = \approx$.

The proof for (i) essentially proceeds by showing $\mathbf{R}\cup\mathbf{id}$ to be a typed bisimulation where $\mathbf{R}$ is inductively generated by the following rules:

$$\begin{array}{rcl} C[\overline{x}(\vec{y})P]|x(\vec{y}).Q & \mathbf{R} & C[(\nu\vec{y})(P|Q)] \\ C[\overline{x}(\vec{y})P]|!x(\vec{y}).Q & \mathbf{R} & C[(\nu\vec{y})(P|Q)]\,|\,!x(\vec{y}).Q \\ (\nu x)!x(\vec{y}).Q & \mathbf{R} & \mathbf{0} \end{array}$$

Here $C[]$ is an arbitrary context not binding x.

To establish (ii), assume that $P \approx Q$. By Lemma 4.2 (i) we can find $\triangleright$-normal forms P_{nf} and Q_{nf} of P and Q respectively such that $P \mapsto^* P_{nf}$ and $Q \mapsto^* Q_{nf}$. Hence by (i) $P_{nf} \approx Q_{nf}$. But Lemma 4.2 (ii) implies that $\approx$ restricted to $\triangleright$-normal forms is contained in $\longleftrightarrow$, hence $P \mapsto^* P_{nf} \longleftrightarrow Q_{nf}$ and $Q \mapsto^* Q_{nf}$ which means $P \longleftrightarrow Q$, as required.

5. Fully Abstract Embedding of $\lambda_{\rightarrow,\times,+}$

5.1. The Functional Calculus

We use the simply typed λ-calculus with products and sums ($\lambda_{\rightarrow,\times,+}$ from now on) as a testbed for the expressiveness of the presented calculus. We have chosen $\lambda_{\rightarrow,\times,+}$ because of its rich type structures and non-trivial equational theory. For simplicity we omit base types other than unit. We review the syntax of types and terms below, with i ranging over $\{1,2\}$.

$$\begin{array}{rcl} T & ::= & \mathtt{unit} \mid T_1 \rightarrow T_2 \mid T_1 \times T_2 \mid T_1 + T_2 \\ M & ::= & x \mid () \mid \lambda x{:}T.M \mid \langle M,N\rangle \mid \pi_i(M) \\ & \mid & \mathtt{in}_i(M) \mid \mathtt{case}\ L\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} \end{array}$$

We write $M \equiv_\alpha N$ for the α-equality on terms. A term is *closed* if no variables occur free.

The reduction relation, written $\rightsquigarrow$, and the typing rules are standard [14, 31]. We write $E \vdash M : T$ when a term M is typable with type T under a base E. We write $C[\,]_T : T'$ for a (typed) context of type T' with one hole of type T. We often omit type annotations from terms and contexts. We write $M \Downarrow N$ when $M \rightsquigarrow^* N$ and $N \not\rightsquigarrow$. A *normal form* is a term which has no further reductions.

Equality in $\lambda_{\rightarrow,\times,+}$ is not as simple as it may look, due to the existence of sums [12]. To have a semantically meaningful equality, we use observation of "values", cf. [31]. Let $\mathtt{true} \stackrel{\text{def}}{=} \mathtt{in}_1(())$ and $\mathtt{false} \stackrel{\text{def}}{=} \mathtt{in}_2(())$, both of type $\mathbb{B} \stackrel{\text{def}}{=} \mathtt{unit} + \mathtt{unit}$. Then $E \vdash M \cong_\lambda N : T$ when, for each context $C[\,]_T : \mathbb{B}$ such that $C[M]$ and $C[N]$ are closed, we have ($C[M] \Downarrow \mathtt{true} \Leftrightarrow C[N] \Downarrow \mathtt{true}$). The same equality is obtained by taking observability at each sum type, justifying all commuting conversions and η-rules.

5.2. The π-Calculus: Extension with Branching

Before the encoding, we extend the typed π-calculus to its full syntax [8] by incorporating branching. Branching is necessary to represent sums in $\lambda_{\rightarrow,\times,+}$ and is also used for defining a reduction-based typed congruence [21, 40].

$$\begin{array}{rcl} P & ::= & \cdots \mid x[\&_i(\vec{y}_i{:}\vec{\tau}_i).P_i] \mid\ !x[\&_i(\vec{y}_i{:}\vec{\tau}_i).P_i] \mid \overline{x}\mathtt{in}_i(\vec{y}{:}\vec{\tau})P \\ \tau_{\mathrm{I}} & ::= & \cdots \mid [\&_i\vec{\tau}_i]^{\downarrow} \mid\ [\&_i\vec{\tau}_i]^{!} \\ \tau_{\mathrm{O}} & ::= & \cdots \mid [\oplus_i\vec{\tau}_i]^{\uparrow} \mid\ [\oplus_i\vec{\tau}_i]^{?} \end{array}$$

The additional reduction rules are defined as:

$$x[\&_i(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_j(\vec{y}_j)\cdot Q \longrightarrow (\nu\vec{y}_j)(P_j \mid Q)$$
$$!x[\&_i(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_j(\vec{y}_j)Q \longrightarrow\ !x[\&_i(\vec{y}_i).P_i] \mid (\nu\vec{y}_j)(P_j \mid Q)$$

Then $\mapsto$ is defined similarly as Definition 3.2. The linear typing rules are given in Appendix A. All arguments and results in the preceding sections carry over to the full syntax without alteration.[2]

Let us say A is *closed* when $\mathsf{md}(A) \subseteq \{!,\bot\}$. Now write $\Gamma \vdash P \Downarrow^i_x$ when $P \Downarrow (\nu\vec{y})(\overline{x}\mathtt{in}_i(\vec{z})P_0 \mid R)$ with $x \notin \{\vec{y}\}$ where $\Gamma \vdash P \triangleright A \otimes \uparrow x$ with A closed. We then define $\cong_{\mathsf{sn}}$ as the maximum typed congruence such that if $\Gamma \vdash P \cong_{\mathsf{sn}} Q$ and $\Gamma \vdash P \Downarrow^i_x$ then $\Gamma \vdash Q \Downarrow^i_x$ (cf. [21, 40]). We use the following two lemma about $\cong_{\mathsf{sn}}$, which is proved as in [8].

Lemma 5.1 (context lemma) *Let $\Gamma \vdash P_j \triangleright A$ ($j = 1,2$) with Γ paired. Then $P_1 \cong_{\mathsf{sn}} P_2$ iff: $P_1 \mid R \Downarrow^i_x \Leftrightarrow P_2 \mid R \Downarrow^i_x$ for each $\Gamma \cdot x : [\oplus_{1,2}]^{\uparrow} \vdash R \triangleright B$ s.t. $A \asymp B$.*

5.3. Embedding and Full Abstraction

The encoding of $\lambda_{\rightarrow,\times,+}$ is given in Figure 4. It adapts Milner's call-by-name encoding [28] to our type structure by adding an indirection at each λ-abstraction. The basic correspondence result follows. Note that in the second statement, there is an exact operational correspondence between $\rightsquigarrow$ and $\mapsto$: $\rightsquigarrow$ is simulated by $\mapsto$ directly, not up to some semantic equality.

Proposition 5.2 *Let $E \vdash M : T$ below with $\mathsf{fn}(E) = \{\vec{y}\}$.*

i. $E^\circ \cdot u : T^\circ \vdash [\![M : T]\!]_u \triangleright\ !u \rightarrow ?\vec{y}$ *is well-typed.*

ii. $M \rightsquigarrow M' \ \Rightarrow\ [\![M]\!]_u \mapsto^+ [\![M']\!]_u$.

Corollary 5.3 $\lambda_{\rightarrow,\times,+}$ *is strongly normalising.*

PROOF: By Theorem 3.9 using: if $[\![N_1]\!]_u \equiv [\![N_2]\!]_u$ with N_i in normal form then $N_1 \equiv_\alpha N_2$. ■

The above corollary offers a faithful *computational embedding* of $\lambda_{\rightarrow,\times,+}$: we now show that this also extends to semantics. First, by Proposition 5.2 and Corollary 5.3:

Lemma 5.4 (computational adequacy) *Let $M : \mathbb{B}$ be closed. Then $M \Downarrow \mathtt{true}$ iff $[\![M]\!]_u \Downarrow_e [\![\mathtt{true}]\!]_u$.*

Corollary 5.5 (soundness) $[\![E \vdash M : T]\!]_u \cong_{\mathsf{sn}} [\![E \vdash N : T]\!]_u$ *implies $E \vdash M \cong_\lambda N : T$.*

[2] A minor change is Proposition 2.3 (i): for a $\uparrow$-channel, "precisely once" becomes, under a branching input, "precisely once in each branch".

$$\textbf{(Type)}\quad \mathtt{unit}^\circ \stackrel{\text{def}}{=} ((\,)^\uparrow)^! \qquad (T_1 \Rightarrow T_2)^\circ \stackrel{\text{def}}{=} (\overline{T_1^\circ}(T_2^\circ)^\uparrow)^! \qquad (T_1 \times T_2)^\circ \stackrel{\text{def}}{=} ((T_1^\circ T_2^\circ)^\uparrow)^! \qquad (T_1 + T_2)^\circ \stackrel{\text{def}}{=} ([T_1^\circ \oplus T_2^\circ]^\uparrow)^!$$

$$\textbf{(Base)}\quad \emptyset^\circ \stackrel{\text{def}}{=} \emptyset \qquad (E \cdot x{:}T)^\circ \stackrel{\text{def}}{=} E^\circ \cdot x{:}\overline{T^\circ}$$

(Terms) (if $T_2 = T_{12} \Rightarrow T_{22}$ then $\vec{z} = z_1 z_2$ else $\vec{z} = z$)

$$[\![x : T]\!]_u \stackrel{\text{def}}{=} [u \to x]^{T^\circ}$$

$$[\![MN : T_2]\!]_u \stackrel{\text{def}}{=} !u(\vec{z}).(\nu mn)([\![M : T_1 \Rightarrow T_2]\!]_m \,|\, [\![N : T_1]\!]_n \,|\, \mathsf{Arg}\langle mn\vec{z}\rangle^{T_1 \Rightarrow T_2})$$

$$[\![\lambda x : T_1.M : T_1 \Rightarrow T_2]\!]_u \stackrel{\text{def}}{=} !u(xz).\overline{z}(m)[\![M : T_2]\!]_m$$

$$[\![\langle M_1, M_2\rangle : T_1 \times T_2]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}(m_1 m_2)([\![M_1 : T_1]\!]_{m_1} \,|\, [\![M_2 : T_2]\!]_{m_2})$$

$$[\![\pi_1\ (M) : T_1]\!]_u \stackrel{\text{def}}{=} !u(\vec{z}).(\nu m)([\![M : T_1 \times T_2]\!]_m \,|\, \mathsf{Proj}_1\langle m\vec{z}\rangle^{T_1})$$

$$[\![() : \mathtt{unit}]\!]_u \stackrel{\text{def}}{=} !u(x).\overline{x}$$

$$[\![\mathtt{inl}(M) : T_1 + T_2]\!]_u \stackrel{\text{def}}{=} !u(c).\overline{c}\mathtt{inl}(m)[\![M : T_1]\!]_m$$

$$[\![\mathtt{case}\ L\ \mathtt{of}\ \mathtt{inl}(x_1)M_1\ \mathtt{or}\ \mathtt{inr}(x_2)M_2 : T]\!]_u \stackrel{\text{def}}{=} !u(\vec{z}).(\nu l)([\![L : T_1 + T_2]\!]_l \,|\, \mathsf{Sum}\langle l\vec{z}, (x_i)M_i\rangle^T)$$

$$\mathsf{Arg}\langle mn\vec{z}\rangle^{T_1 \Rightarrow T_2} \stackrel{\text{def}}{=} \overline{m}(n'c').([n' \to n]^{T_1^\circ} \mid c'(u').\mathsf{Con}\langle u'\vec{z}\rangle^{T_2^\circ})$$

$$\mathsf{Proj}_i\langle m\vec{z}\rangle^T \stackrel{\text{def}}{=} \overline{m}(e)e(v_1 v_2).\mathsf{Con}\langle v_i\vec{z}\rangle^{T^\circ}$$

$$\mathsf{Sum}\langle l\vec{z}, (x_i)M_i\rangle^T \stackrel{\text{def}}{=} \overline{l}(c)c[\&_{1,2}(x_i).(\nu m)([\![M_i : T]\!]_m \,|\, \mathsf{Con}\langle m\vec{z}\rangle^{T^\circ})]$$

$$\mathsf{Con}\langle x\vec{y}\rangle^{(\vec{\tau})^!} \stackrel{\text{def}}{=} \overline{x}(\vec{y}')\textstyle\prod[y_i' \to y_i]^{\overline{\tau_i}}$$

$$[x \to x']^{\&_i(\vec{\tau}_i)^\downarrow} \stackrel{\text{def}}{=} x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$$

$$[x \to x']^{\&_i(\vec{\tau}_i)^!} \stackrel{\text{def}}{=} !x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$$

We omit $\mathtt{inr}(M)$ and $\pi_2(M)$. For the copy-cats of unary types we assume the indexing sets to be singletons.

Figure 4. Encoding of $\lambda_{\to,\times,+}$

For completeness, we use a specific class of linear processes. Let us say P is *sequential* [8] if it is typable by the same system as Figures 2 and rules in Appendix A, augmented with the sequentiality constraint in Figure 1 of [8]. A key lemma for completeness follows.

Lemma 5.6 (sequential testability) *Let* $E \stackrel{def}{=} \vec{y}{:}\vec{S}$ *and* $E^\circ \cdot u : T^\circ \vdash P_n \triangleright\, !u \otimes ?\vec{y}\ (n = 1,2)$. *Then* $P_1 \cong_{\mathsf{sn}} P_2$ *iff:*

$$(\Pi_j R_j \,|\, P_1 \,|\, Q) \Downarrow_\chi^i \quad \Leftrightarrow \quad (\Pi_j R_j \,|\, P_2 \,|\, Q) \Downarrow_\chi^i \qquad (i = 1,2)$$

for each sequential $y_j : S_j^\circ \vdash R_j \triangleright\, !y_j$ *and sequential* $u : \overline{T^\circ}, x : [\oplus_{1,2}]^\uparrow \vdash Q \triangleright\, ?u \otimes \uparrow x$.

For the proof, we use Lemma 5.1, and by assuming, via Theorem 3.9, the context to be in NF_e, we obtain $\Pi_j R_j$ and Q of desired types.

The final step is to show that each process with $\lambda_{\to,\times,+}$-types is translatable to a *canonical normal form* [4, 22] (CNF) whose grammar is given below.

$$\begin{array}{rcl} F & ::= & () \mid x \mid \lambda x.F \mid \langle F_1, F_2\rangle \mid \mathtt{in}_i(F) \mid \\ & & \mathtt{let}\ () = z\ \mathtt{in}\ F \mid \mathtt{let}\ x = zF\ \mathtt{in}\ F' \\ & & \mathtt{let}\ \langle x, y\rangle = z\ \mathtt{in}\ F \mid \mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{in}_i(x_i).F_i\} \end{array}$$

We omit the typing and reduction rules. CNFs are translated to $\lambda_{\to,\times,+}$-terms in the standard way without changing their compositional behaviour, which we write F° (see [41] for definition). The range of this map exhausts all normal forms of $\lambda_{\to,\times,+}$. We can now prove:

Proposition 5.7 (definability) *Let* $E^\circ \cdot u : T^\circ \vdash P \triangleright\, !u \to ?\vec{y}$ *sequential with* $\mathsf{fn}(E) = \{\vec{y}\}$. *Then* $P \cong_{\mathsf{sn}} [\![F^\circ]\!]_u$ *for some* $E \vdash F : T$

The proof is by induction on the size of sequential processes under all $\lambda_{\to,\times,+}$-bases and types. We can now establish full abstraction.

Theorem 5.8 (full abstraction) $E \vdash M_1 \cong_\lambda M_2 : T$ *iff* $E^\circ \cdot u : T^\circ \vdash [\![M_1 : T]\!]_u \cong_{\mathsf{sn}} [\![M_2 : T]\!]_u$.

PROOF: By Corollary 5.5, we only have to show the "then" direction. Suppose $M_1 \cong_\lambda M_2$ but $[\![M_1]\!]_u \not\cong_{\mathsf{sn}} [\![M_2]\!]_u$. By the latter, take ΠR_j and Q as in Lemma 5.6 s.t. $(\nu\vec{y}u)(\Pi R_j \,|\, [\![M_i]\!]_u \,|\, Q') \Downarrow_e [\![\mathbf{b}_i]\!]_w$ $(i = 1,2)$ with $Q' = !w(x).Q$, $\mathbf{b}_1 = \mathtt{true}$ and $\mathbf{b}_2 = \mathtt{false}$. By Proposition 5.7, we have $\vec{F}$ and F' s.t. $(\lambda\vec{y}.F'^\circ x)M_i\vec{F}^\circ \Downarrow \mathbf{b}_i$ $(i = 1,2)$, which contradicts Lemma 5.4, hence done. ■

6. Discussion and Further Work

Summary The present study is part of our quest to articulate significant classes of computational behaviour using typed π-calculi. Previous work [8] introduced *affine, sequential* types for the π-calculus and established full abstraction for an encoding of PCF. Using causality between names, the present text refines affine, sequential types into *linear* types to ensure strong normalisability and full abstraction for $\lambda_{\to,\times,+}$. Figure 5 shows the relationship between these results.

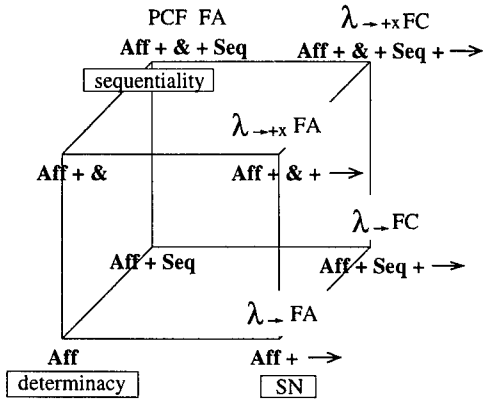


Figure 5. A Family of Affine/Linear Systems

- The addition of branching types is indicated by &, $\rightarrow$ adds causality to action types, and Seq stands for the inclusion of the sequentiality constraints used in [8].
- Determinacy, SN and sequentiality are properties guaranteed by each typing system.
- FC denotes *full completeness* of the embedding of the corresponding λ-calculus into the π-calculus (in the sense of [3]), while FA stands for *full abstraction*.

For example, the linear typing system in § 2 corresponds to Aff + $\rightarrow$, its branching extension in §5 to Aff + & + $\rightarrow$ and the sequential system in [8] to Aff + & + Seq. Note also that the development in § 5 shows that our encoding is already 'almost' fully complete intensionally and indeed becomes fully complete by quotienting with the observational congruence. It is also notable that we could have used the call-by-value encoding in [28] to obtain exactly the same result, indicating the flexibility of the proposed calculus to encode functional SN behaviour.

Liveness in Interaction A consequence of strong normalisability is liveness in interaction: if a typed agent calls another replicated typed agent and waits for its answer at a truly linear channel x, then an answer is guaranteed to eventually arrive at x, however complex intermediate interaction sequences would be. Below see § 5.2 for the notion of closed action types.

Proposition 6.1 (linear honesty) *Let* $\Gamma \vdash x:(\tau)^!$ *be such that* $\mathsf{md}(\tau) = \uparrow$. *Suppose* $\Gamma \vdash P \triangleright A$ *with A closed. Then* $P \xrightarrow{x(y)} P'$ *implies* $P' \longrightarrow^* \xrightarrow{l}$ *where l is an output at y.*

We can strengthen Proposition 6.1 by incorporating the possibility that the client itself interacts with the server towards the eventual answer [18]. The central point of the present liveness property is that, in spite of such nested, complex webs of procedure calls, each client is still guaranteed to receive an answer, strengthening preceding related type disciplines, cf. [24, 25, 40].

State and Non-functional Control It is an important subject of study to extend our typing system to allow incorporation of state and non-functional control. The resulting calculi would be useful as a theoretical basis for the application of SN in a wider realm. Such a formalism might also be useful as a meta-language for logical systems with e.g. non-deterministic cut elimination procedures.

So far we have verified that our proof method is also applicable to SN for first-order stateful processes, albeit under a sequentiality constraint [8]. We foresee no fundamental difficulty in extending the results to concurrent stateful computation, although the lack of the Church-Rosser property would make reasoning harder.

Complex Causality The present work adds minimum causality to the system in [8] to ensure SN of replicated processes. However, our SN proof seems to be able to cope, without significant change, with more complex causality relations: for example, we could relax the channel type constraints and extend action types to finite graph structures between arbitrary linear nodes as in [40]. An even wider class of SN interactions would be typable if we further allowed edges of the more general form $px \rightarrow qy$, where $p \in \{\downarrow, \uparrow, ?\}$ and $q \in \{!, \downarrow, \uparrow\}$ (i.e. replicated and linear nodes can be mixed). Diverse structures would be embeddable in such an extension, including full proof nets [7]. The status of strong reduction would become subtle in this setting, cf. [12].

Second-order and Other Extensions Can the presented results be augmented to cover more expressive notions of types studied in functional calculi? Adding recursive types [29, 39] easily leads to a system that is not strongly normalising: for example, the encoding, following Figure 4, of $(\lambda x.xx)(\lambda x.xx)$ would be typable. Regarding second-order types, our recent work [9] demonstrates that such extensions coexist harmoniously with SN, as they do in the corresponding functional calculi. In particular, the causality constraints formalised in the present paper are sufficient to encode System F fully abstractly in the second-order extension of the present system. Other, more refined type structures would also be worth studying in the present context: the π-calculus offers a natural habitat to SN typing systems for stateful, interactive and mobile computation.

Game Semantics In game semantics, "winning strategies" represent strong normalisation [3]. This representation ensures, essentially by definition, that composition of two winning strategies will never go into infinite τ-actions (which would make the strategy partial). This extensional

representation of SN does not directly suggest concrete type disciplines to ensure SN for mobile processes (although the liveness property discussed in Proposition 6.1 closely corresponds to the games-based characterisation of SN). On the other hand, the present work may offer new ways to formulate the notion of SN in game semantics, where acyclicity conditions are explicitly incorporated into game types.

References

[1] Abramsky, S., Computational interpretation of linear logic. *TCS*, Vol. 111 (1993) 3–57, 1993.

[2] Abramsky, S., Proofs as Processes, *TCS*, Vol. 135 (1994) 5–9, 1994.

[3] Abramsky, S. and Jagadeesan, R., Games and Full Completeness for Multiplicative Linear Logic. *JSL*, Vol. 59, 1994.

[4] Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF, *Info. & Comp.* 163 (2000), 409-470.

[5] Abramsky, S., Process Realizability, A Tutorial Workshop on Realizability Semantics and Applications, 1999. Available at web.comlab.ox.ac.uk/oucl/work/samson.abramsky.

[6] Barendregt, H., Lambda Calculi with Types, Handbook of Logic in Computer Science, Vol 2, 118–310, Oxford, 1992.

[7] Bellin, G., and Scott, P.J., On the Pi-calculus and linear logic, *TCS*, Vol. 135, 11-65, 1994.

[8] Berger, M., Honda, K. and Yoshida, N., Sequentiality and the π-Calculus, to appear in *Proc. TLCA01*, LNCS, Springer, 2001. Available at www.dcs.qmw.ac.uk/~kohei.

[9] Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus. To appear as a CS technical report, Queen Mary. Available at www.dcs.qmw.ac.uk/~martinb.

[10] Boudol, G., Asynchrony and the pi-calculus, INRIA Research Report 1702, 1992.

[11] Gallier, J. H., On Girard's "Candidats de Reductibilité", 123–203, Logic and Computer Science, Academic Press Limited, 1990.

[12] Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, 1–102, 1987.

[13] Girard, J.-Y., Lafont Y. and Taylor, P., *Proofs and Types*, vol. 7 of Cambridge Tracts in Theoretical Computer Science, CUP, 1989.

[14] Gunter, C.A., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.

[15] Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A. and Nettles, S. PLAN: A Packet Language for Active Networks, *Proc. ICFP'98*, cf. [33].

[16] Honda, K., Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, 509-523, 1993.

[17] Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.

[18] Honda, K., Kubo, M. and Vasconcelos, V., Language Primitives and Type Discipline for Structured Communication-Based Programming. *ESOP'98*, LNCS 1381, 122–138. Springer-Verlag, 1998.

[19] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, 133–147, Springer-Verlag 1991.

[20] Honda, K. Vasconcelos, V., and Yoshida, N. Secure Information Flow as Typed Process Behaviour, *ESOP '00*, LNCS 1782, 180–199, Springer-Verlag, 2000.

[21] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *TCS*, 437–486, Vol. 151, North-Holland, 1995.

[22] Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285-408.

[23] Igarashi, A. and Kobayashi, N., A generic type system for the pi-calculus, *POPL'01*, ACM, 2001.

[24] Kobayashi, N., A partially deadlock-free typed process calculus, *ACM TOPLAS*, Vol. 20, No. 2, 436–482, 1998.

[25] Kobayashi, N., Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness, *Proc. of TCS2000*, LNCS 1872, 365-389, Springer, 2000.

[26] Kobayashi, N., Pierce, B., and Turner, D., Linear Types and π-calculus, *POPL'96*, 358–371, ACM Press, 1996.

[27] Lafont, Y., Interaction Nets, *POPL'90*, 95–108, ACM Press, 1990.

[28] Milner, R., Functions as Processes. *MSCS*, 2(2), 119–146, CUP, 1992.

[29] Milner, R., Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.

[30] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes, *Info. & Comp.* 100(1), 1–77, 1992.

[31] Mitchell, J. *Foundations for Programming Languages*, MIT Press, 1996.

[32] Pierce, B.C. and Sangiorgi, D, Typing and subtyping for mobile processes. *LICS'93*, 187–215, IEEE, 1993.

[33] PLAN: A Packet Language for Active Networks, SwitchWare Project, University of Pennsylvania, available from www.cis.upenn.edu/~switchware/.

[34] Quaglia, P. and Walker, D., On Synchronous and Asynchronous Mobile Processes, *FoSSaCS 00*, LNCS 1784, 283–296, Springer, 2000.

[35] Sangiorgi, D. π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, North-Holland, 1996.

[36] Sangiorgi, D., The name discipline of uniform receptiveness, *ICALP'97*, LNCS 1256, 303–313, Springer, 1997.

[37] Tait, W., Intensional interpretation of functionals of finite type, I. *J. Symb. Log*, 32, 198–212, 1967.

[38] Vasconcelos, V., Typed concurrent objects. *ECOOP'94*, LNCS 821, 100–117. Springer, 1994.

[39] Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, 524–538, Springer-Verlag, 1993.

[40] Yoshida, N., Graph Types for Monadic Mobile Processes, *FST/TCS'16*, LNCS 1180, 371–387, Springer-Verlag, 1996. Full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.

[41] The full version of this paper, MCS technical report, 2001-09, University of Leicester, March, 2001. Available at: www.mcs.le.ac.uk/~nyoshida/paper.html.

A. Appendix: Typing Rules for Branching

$$\frac{\Gamma \vdash x : [\&_i \vec{\tau}_i]^{\uparrow} \quad \Gamma \cdot \vec{y}_i : \vec{\tau}_i \vdash P_i \triangleright C_i^{\neg x} \quad (C_i/\vec{y}_i = ?B)}{\Gamma \vdash x[\&_i(\vec{y}_i : \vec{\tau}_i).P_i] \triangleright\, !x \to B} \ (\mathsf{Bra}^{\uparrow})$$

$$\frac{\Gamma \vdash x : [\oplus_i \vec{\tau}_i]^{?} \quad \Gamma \cdot \vec{y}_i : \vec{\tau}_i \vdash P \triangleright C \quad (C_i/\vec{y}_i = A \rightarrowtail ?x)}{\Gamma \vdash \overline{x}\mathtt{in}(\vec{y}_i : \vec{\tau}_i)P \triangleright A \odot ?x} \ (\mathsf{Sel}^{?})$$

($\mathsf{Bra}^{\downarrow}$) and ($\mathsf{Sel}^{\uparrow}$) are defined similarly.

A symbolic labelled transition system for coinductive subtyping of $F_{\mu\leq}$ types

Alan Jeffrey
DePaul University

Extended Abstract

Abstract. *$F_{\leq}$ is a typed λ-calculus with subtyping and bounded polymorphism. Typechecking for $F_{\leq}$ is known to be undecidable, because the subtyping relation on types is undecidable. $F_{\mu\leq}$ is an extension of $F_{\leq}$ with recursive types. In this paper, we show how symbolic labelled transition system techniques from concurrency theory can be used to reason about subtyping for $F_{\mu\leq}$. We provide a symbolic labelled transition system for $F_{\mu\leq}$ types, together with an an appropriate notion of simulation, which coincides with the existing coinductive definition of subtyping. We then provide a 'simulation up to' technique for proving subtyping, for which there is a simple model checking algorithm. The algorithm is more powerful than the usual one for $F_{\leq}$, for example it terminates on Ghelli's canonical example of nontermination.*

1 Introduction

Symbolic labelled transition systems [11] have been used in concurrency theory to provide finite-state representations of infinite systems. They have been used to model-check systems with data dependencies, where the nïave state space exploration technique would produce an infinite state space, and so not terminate.

In this paper, we apply symbolic lts techniques to a new problem area: that of deciding subtyping for polymorphic λ-calculi.

Subtyping and polymorphism. Curien and Ghelli's [5] $F_{\leq}$ is a typed λ-calculus with bounded polymorphism and subtyping. It is based on Bruce and Longo's [2] development of Cardelli and Wegner's [3] *Fun* language.

The most interesting rule in $F_{\leq}$ is that for subtyping of polymorphic types:

$$\frac{\Gamma \vdash T_2 \leq T_1 \quad \Gamma.X \leq T_2 \vdash U_1 \leq U_2}{\Gamma \vdash (\forall X \leq T_1 . U_1) \leq (\forall X \leq T_2 . U_2)} \text{ (Full } F_{\leq})$$

This is a stronger rule than the rule used in *Fun*, which is:

$$\frac{\Gamma.X \leq T \vdash U_1 \leq U_2}{\Gamma \vdash (\forall X \leq T . U_1) \leq (\forall X \leq T . U_2)} \text{ (Kernel } F_{\leq})$$

It is routine to develop an algorithm to check the subtyping property of Kernel $F_{\leq}$, but subtyping for Full $F_{\leq}$ has turned out to be surprisingly complex. Curien and Ghelli [5] gave an algorithm for checking subtyping, with a correctness proof provided by Ghelli [7]. Later, Ghelli [9] showed that this algorithm is not guaranteed to terminate. Pierce [14] showed that Ghelli's example of nontermination can be generalized to code a Turing machine, and so subtyping (and hence typechecking) for $F_{\leq}$ is undecidable.

Subtyping and recursive types. Recursive types are a common programming language feature, typified by ML's `datatype` construct. Amadio and Cardelli [17] investigated the relationship between subtyping and recursive types. Brand and Henglein [1] reformulated subtyping in terms of coinductive relations on types, which we will use here. The coinductive presentation of type systems for subtyping in the presence of recursive types has been used by Pierce and Sangiorgi [16] for the π-calculus, Turner [20] for Pict and Sewell [19] for a distributed π-calculus. A good introduction is by Gapeyev, Levin and Pierce [6].

Ghelli [8] has investigated the relationship between subtyping, recursive types and polymorphic types, in the recursive extension to $F_{\leq}$, called $F_{\mu\leq}$. He has shown a number of surprising results: adding recursion to $F_{\leq}$ is not conservative, and $F_{\mu\leq}$ does not satisfy the transitivity elimination property. These results are for the inductive definition of subtyping, however, where here we look at the coinductive definition, which is much better behaved. Colazzo and Ghelli have provided an algorithm for deciding subtyping of Kernel $F_{\mu\leq}$ [4]: much of this paper is based on that algorithm.

Symbolic labelled transition systems. Labelled transition systems are a form of nondeterministic automaton, where all states are considered to be accepting states. They were proposed by Milner [12, 13] as an appropriate model for concurrent systems. They have since been used to model higher-order computation, for example Gordon's [10] lts model of the simply-typed λ-calculus.

0-7695-1281-X/01 $10.00 © 2001 IEEE

One problem with lts models is that they can produce infinite models of systems which should be finite-state. For example, the process defined:

$$P = \mathsf{in}\ (x : \mathsf{int}); \mathsf{out}\ (x+1); P$$

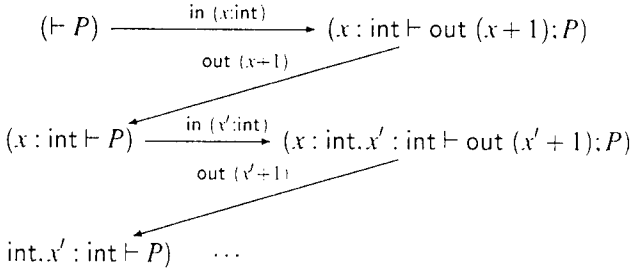
has transitions:

$$(P) \xrightarrow{\mathsf{in}\ (n)} (\mathsf{out}\ (n+1); P) \xrightarrow{\mathsf{out}\ (n+1)} (P)$$

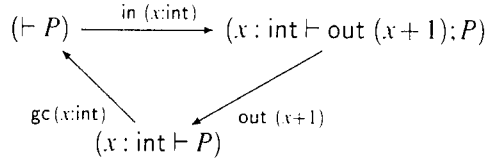
for every integer n and so is infinite-state. Hennessy and Lin [11] proposed using *symbolic* labelled transition systems as an appropriate finitary representation. A symbolic lts includes free variables, so rather than having nodes being closed processes, and edges labelled with closed expressions, the nodes are processes together with their free variables, and the edges are labelled with open expressions. For example:

$$(\vdash P) \xrightarrow{\mathsf{in}\ (x:\mathsf{int})} (x : \mathsf{int} \vdash \mathsf{out}\ (x+1); P) \xrightarrow{\mathsf{out}\ (x+1)} (x : \mathsf{int} \vdash P)$$

Unfortunately, this system is still infinite-state, since the context can grow unboundedly:



For this reason, symbolic techniques often work 'up to garbage collection' where unneeded free variables can be removed from the context. For example, the above process can be given a finite symbolic representation as:



Symbolic lts's have been used to provide finite-state representations of systems that would otherwise be infinite-state.

Contributions of this paper. In this paper, we apply the techniques of symbolic labelled transition systems to the problem of subtyping $F_{\mu\leq}$. In particular, we:

- Give an alternative characterization of subtyping for $F_{\mu\leq}$, as *polar simulation* for an appropriate symbolic lts.
- Use a variant of Milner and Sangiorgi's [18] *bisimulation up to* method to give a sound proof technique for subtyping.
- Provide an algorithm for finding an appropriate polar simulation, if one exists.
- Show that the algorithm is partially correct: if it terminates, it does so with the right answer.
- Show that the algorithm is strictly more powerful than the standard algorithm for $F_{\leq}$, and at least as powerful as Colazzo and Ghelli's algorithm for Kernel $F_{\mu\leq}$.

Acknowledgements. I would like to thank Benjamin Pierce, James Riely and Peter Sewell for useful discussions about this material. Donald Knuth's TEX typesetting system, Leslie Lamport *et al.*'s LATEX document markup language, and Paul Taylor's diagrams package were used in the preparation of this paper.

2 The type system of $F_{\mu\leq}$

In this section, we review the types system used in Ghelli's [8] $F_{\mu\leq}$. There are some minor syntactic differences between the types presented here and Ghelli's, but they are equally expressive. We have added type constants such as int and real to the language, to make examples clearer, they are not required for any of the technical development.

Let K range over a finite collection of type constants, such as int and real. The syntax of types is given:

$$T.U.V \quad ::= \quad T \to U \mid \mathsf{Top} \mid K \mid \forall X \leq T.U \mid \mu^{-} X.T \mid X$$

Define the *free variables* of a type as:

$$\mathsf{fv}(T) \quad = \quad \mathsf{fv}^{+}(T) \cup \mathsf{fv}^{-}(T)$$

where the *polarized free variables* are:

$$\begin{aligned}
\mathsf{fv}^{=}(T \to U) &= \mathsf{fv}^{\mp}(T) \cup \mathsf{fv}^{\pm}(U) \\
\mathsf{fv}^{=}(\mathsf{Top}) &= \emptyset \\
\mathsf{fv}^{\pm}(K) &= \emptyset \\
\mathsf{fv}^{\mp}(\forall X \leq T.U) &= \mathsf{fv}^{\mp}(T) \cup (\mathsf{fv}^{\pm}(U) \setminus \{X\}) \\
\mathsf{fv}^{\pm}(\mu^{-} X.T) &= \mathsf{fv}^{\pm}(T) \setminus \{X\} \\
\mathsf{fv}^{+}(X) &= \{X\} \\
\mathsf{fv}^{-}(X) &= \emptyset
\end{aligned}$$

A *type context* is a sequence of variables with type bounds:

$$\Gamma.\Delta \quad ::= \quad X_1 \leq T_1, \ldots, X_n \leq T_n$$

where we ignore the order of bindings. The *domain* of a context $\mathsf{dom}(\Gamma)$ is defined:

$$\mathsf{dom}(X_1 \leq T_1, \ldots, X_n \leq T_n) \quad = \quad \{X_1, \ldots, X_n\}$$

When $X \in \mathsf{dom}(\Gamma)$ we define $\Gamma(X)$ as:

$$(\Gamma, X \leq T)(X) \quad = \quad T$$

The *well-formed context* judgment $\Gamma \vdash \diamond$ is defined:

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{\Gamma \vdash T}{\Gamma, X \leq T \vdash \diamond}[X \notin \mathsf{dom}(\Gamma)]$$

where the *well-formed type* judgment $\Gamma \vdash T$ is defined:

$$\frac{\Gamma \vdash T \quad \Gamma \vdash U}{\Gamma \vdash T \to U} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Top}} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash K} \qquad \frac{\Gamma, X \leq T \vdash U}{\Gamma \vdash \forall X \leq T . U}$$

$$\frac{\Gamma, X \leq T \vdash \diamond}{\Gamma, X \leq T \vdash X} \qquad \frac{\Gamma, X \leq \mathsf{Top} \vdash T}{\Gamma \vdash \mu^+ X . T}[X \notin \mathsf{fv}^-(T), T \neq Y]$$

Note that we have required X to occur positively in T in any recursive type $\mu^+ X . T$, and that we cannot form recursive types of the form $\mu^+ X . Y$. These restrictions do not limit the expressive power of the type system, since for any $T(X)$ we can find $T'(X, X')$ such that:

$$T(X) = T'(X, X)$$
$$X \notin \mathsf{fv}^-(T'(X, X')) \quad X' \notin \mathsf{fv}^+(T'(X, X'))$$

then we can define:

$$\mu X . T(X) \quad = \quad \mu^+ X_1 . T'(X_1, \mu^+ X_2 . T'(X_2, X_1))$$

and we can give a greatest fixed point semantics for $\mu X . T$ as:

$$\mu X . Y \quad = \quad \begin{cases} \mathsf{Top} & \text{if } X = Y \\ Y & \text{otherwise} \end{cases}$$

We define α-equivalence on well-formed types as (when $Y \notin \mathsf{dom}(\Gamma)$):

$$(\Gamma, X \leq U \vdash T) \overset{Y/X}{=\!=} (\Gamma[Y/X], Y \leq U \vdash T[Y/X])$$

We assume an ordering $K_1 \leq K_2$ on type constants, for example $\mathsf{int} \leq \mathsf{real}$. This is extended to an *inductive subtyping* judgment $\Gamma \vdash T_1 \leq T_2$ defined:

$$\frac{}{\Gamma \vdash T \leq T} \qquad \frac{\Gamma \vdash T_2 \leq T_1 \quad \Gamma \vdash U_1 \leq U_2}{\Gamma \vdash (T_1 \to U_1) \leq (T_2 \to U_2)}$$

$$\frac{}{\Gamma \vdash T \leq \mathsf{Top}} \qquad \frac{K_1 \leq K_2}{\Gamma \vdash K_1 \leq K_2}$$

$$\frac{\Gamma \vdash T_2 \leq T_1 \quad \Gamma, X \leq T_2 \vdash U_1 \leq U_2}{\Gamma \vdash (\forall X \leq T_1 . U_1) \leq (\forall X \leq T_2 . U_2)} \qquad \frac{\Gamma \vdash \Gamma(X) \leq T}{\Gamma \vdash X \leq T}$$

$$\frac{\Gamma \vdash T_1[(\mu^+ X . T_1)/X] \leq T_2}{\Gamma \vdash (\mu^+ X . T_1) \leq T_2} \qquad \frac{\Gamma \vdash T_1 \leq T_2[(\mu^+ X . T_2)/X]}{\Gamma \vdash T_1 \leq (\mu^+ X . T_2)}$$

A *well-formed relation on types* $\mathcal{R}$ is a relation $\mathcal{R}$ on well-formed types $\Gamma \vdash T$ such that if $(\Gamma_1 \vdash T_1)\ \mathcal{R}\ (\Gamma_2 \vdash T_2)$ then $\Gamma_1 = \Gamma_2$. We shall often write $\Gamma \vDash T_1\ \mathcal{R}\ T_2$ when $(\Gamma \vdash T_1)\ \mathcal{R}\ (\Gamma \vdash T_2)$. For example, the inductive subtyping relation $\leq$ gives a well-formed relation on types:

$$\Gamma \vDash T \leq U \quad \text{iff} \quad \Gamma \vdash T \leq U$$

We regard well-formed relations on types up to α-equivalence, so we can complete the diagram:

$$\begin{array}{ccc} (\Gamma \vdash T) & \overset{\mathcal{R}}{\longleftrightarrow} & (\Gamma \vdash U) \\ {\scriptstyle Y/X}\Big\| & & \Big\|{\scriptstyle Y/X} \\ (\Gamma' \vdash T') & & (\Gamma' \vdash U') \end{array} \quad \text{as} \quad \begin{array}{ccc} (\Gamma \vdash T) & \overset{\mathcal{R}}{\longleftrightarrow} & (\Gamma \vdash U) \\ {\scriptstyle Y/X}\Big\| & & \Big\|{\scriptstyle Y/X} \\ (\Gamma' \vdash T') & \overset{\mathcal{R}}{\longleftrightarrow} & (\Gamma' \vdash U') \end{array}$$

A well-formed relation on types $\mathcal{R}$ is *sound for subtyping* if, for every instantiated subtyping rule:

$$\frac{\Gamma_1 \vdash T_1 \leq U_1 \quad \cdots \quad \Gamma_n \vdash T_n \leq U_n}{\Gamma \vdash T \leq U}$$

we have:

$$\text{if } \Gamma_1 \vDash T_1\ \mathcal{R}\ U_1 \text{ and } \ldots \text{ and } \Gamma_n \vDash T_n\ \mathcal{R}\ U_n \text{ then } \Gamma \vDash T\ \mathcal{R}\ U$$

A well-formed relation on types $\mathcal{R}$ is *consistent with subtyping* if it is sound for subtyping, and whenever $\Gamma \vDash T\ \mathcal{R}\ U$ we can find an instantiated subtyping rule:

$$\frac{\Gamma_1 \vdash T_1 \leq U_1 \quad \cdots \quad \Gamma_n \vdash T_n \leq U_n}{\Gamma \vdash T \leq U}$$

such that:

$$\Gamma_1 \vDash T_1\ \mathcal{R}\ U_1 \text{ and } \ldots \text{ and } \Gamma_n \vDash T_n\ \mathcal{R}\ U_n$$

Let the *coinductive subtyping* relation $\sqsubseteq$ be the largest relation consistent with subtyping.

Proposition 1 *$\leq$ is the smallest relation consistent with subtyping, and so $\leq \subseteq \sqsubseteq$.*

3 Motivation for the symbolic lts semantics for $F_{\mu\leq}$

This paper provides an alternative characterization of subtyping for $F_{\mu\leq}$, using a symbolic labelled transition system. By recasting coinductive subtyping as an lts, it is possible to use existing tools from concurrency theory, notably Milner and Sangiorgi's *bisimulation up to* technique.

The lts has well-formed types as nodes, and edges which reflect the structure of the type. For example, the Top type has no transitions:

$$(\Gamma \vdash \mathsf{Top}) \overset{\alpha}{\not\longrightarrow} (\Gamma' \vdash T')$$

and the type constants have transitions with their name:

$$(\Gamma \vdash \mathsf{int}) \xrightarrow{\mathsf{int}} (\Gamma \vdash \mathsf{Top}) \quad (\Gamma \vdash \mathsf{real}) \xrightarrow{\mathsf{real}} (\Gamma \vdash \mathsf{Top})$$

We can think of the subtyping relation as a *simulation* [13] relation: if T is a supertype of U then any transition of T must have a matching transition from U. For example we can complete the following diagram:

$$\begin{array}{ccc} (\vdash \mathsf{real}) & \overset{\gtrsim}{\leftarrow} & (\vdash \mathsf{int}) \\ {\scriptstyle \mathsf{real}}\downarrow & & \\ (\vdash \mathsf{Top}) & & \end{array} \quad \text{as} \quad \begin{array}{ccc} (\vdash \mathsf{real}) & \overset{\gtrsim}{\leftarrow} & (\vdash \mathsf{int}) \\ {\scriptstyle \mathsf{real}}\downarrow & & \Downarrow{\scriptstyle \widehat{\mathsf{real}}} \\ (\vdash \mathsf{Top}) & \overset{\gtrsim}{\leftarrow} & (\vdash \mathsf{Top}) \end{array}$$

We define the 'matching transition relation' $\xRightarrow{\hat{\alpha}}$ formally in Section 4, for the moment we will just say that it includes $\xrightarrow{\alpha}$, but also includes:

$$(\Gamma \vdash \mathsf{int}) \xRightarrow{\widehat{\mathsf{real}}} (\Gamma \vdash \mathsf{Top})$$

This notion of a 'matching transition relation' is standard in process calculi, where it is used to define weak bisimulation [13]. In general, a *simulation* $\gtrsim$ is a well-formed relation on types where we can complete the diagram:

$$\begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\gtrsim}{\leftarrow} & (\Gamma \vdash T_2) \\ {\scriptstyle \alpha}\downarrow & & \\ (\Gamma' \vdash T_1') & & \end{array} \quad \text{as} \quad \begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\gtrsim}{\leftarrow} & (\Gamma \vdash T_2) \\ {\scriptstyle \alpha}\downarrow & & \Downarrow{\scriptstyle \hat{\alpha}} \\ (\Gamma' \vdash T_1') & \overset{\gtrsim}{\leftarrow} & (\Gamma' \vdash T_2') \end{array}$$

Function types have domain and codomain transitions:

$$\begin{array}{ccc} & (\Gamma \vdash T \to U) & \\ {\scriptstyle \mathsf{dom}}\swarrow & & \searrow{\scriptstyle \mathsf{cod}} \\ (\Gamma \vdash T) & & (\Gamma \vdash U) \end{array}$$

Since function types are contravariant in their first argument and covariant in their second argument, we introduce *polarity* to labels: dom is negative polarity, and cod is positive polarity. This is important when we consider the subtyping relation, for example:

$$\begin{array}{ccccc} (\vdash \mathsf{int} \to \mathsf{real}) & & \overset{\gtrsim}{\longleftarrow} & & (\vdash \mathsf{real} \to \mathsf{int}) \\ {\scriptstyle \mathsf{cod}}\downarrow & {\scriptstyle \mathsf{dom}}\searrow & & {\scriptstyle \widehat{\mathsf{dom}}}\Downarrow & \Downarrow{\scriptstyle \widehat{\mathsf{cod}}} \\ & (\vdash \mathsf{int}) & \overset{\lesssim}{\leftarrow} & (\vdash \mathsf{real}) & \\ (\vdash \mathsf{real}) & & \overset{\gtrsim}{\longleftarrow} & & (\vdash \mathsf{int}) \end{array}$$

Note that after a dom transition, the subtyping relation is inverted, but after a cod transition, it is not. A well-formed relation $\gtrsim$ is a *polar simulation* if it acts as a simulation on positive labels, and on negative labels we can complete the diagram:

$$\begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\gtrsim}{\leftarrow} & (\Gamma \vdash T_2) \\ {\scriptstyle \alpha^-}\downarrow & & \\ (\Gamma' \vdash T_1') & & \end{array} \quad \text{as} \quad \begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\gtrsim}{\leftarrow} & (\Gamma \vdash T_2) \\ {\scriptstyle \alpha^-}\downarrow & & \Downarrow{\scriptstyle \widehat{\alpha^-}} \\ (\Gamma' \vdash T_1') & \overset{\lesssim}{\leftarrow} & (\Gamma' \vdash T_2') \end{array}$$

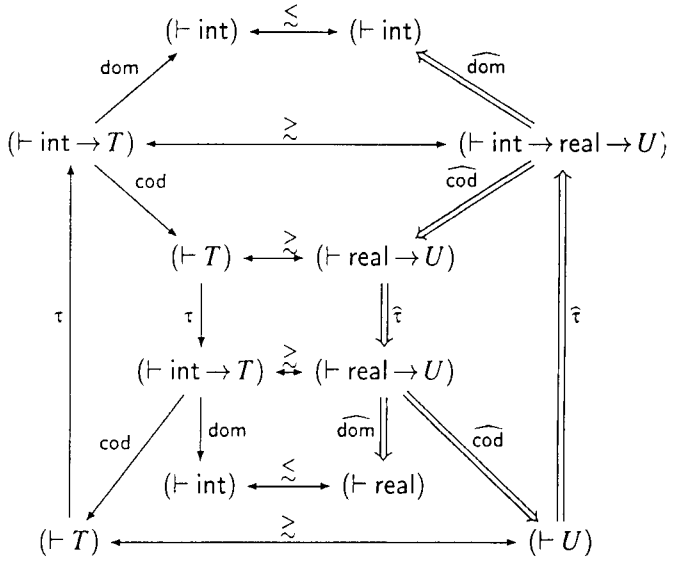
To cope with recursive types, we allow *silent actions* τ, where recursive types can silently unwind:

$$(\Gamma \vdash \mu^+ X.T) \xrightarrow{\tau} (\Gamma \vdash T[\mu^+ X.T/X])$$

For example, if we define:

$$T = \mu^+ X.\, \mathsf{int} \to X \qquad U = \mu^+ Y.\, \mathsf{int} \to \mathsf{real} \to Y$$

then we have a polar simulation for $T \gtrsim U$, since we define the matching transition relation to ignore τ actions:



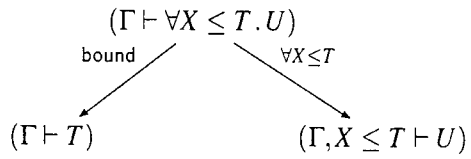
Since we are giving a semantics for types with free variables, we need to give variables transitions: they can either announce themselves, or behave like their bound:

$$\begin{array}{ccc} & (\Gamma \vdash X) & \\ {\scriptstyle X}\swarrow & & \searrow{\scriptstyle \tau} \\ (\Gamma \vdash \mathsf{Top}) & & (\Gamma \vdash \Gamma(X)) \end{array}$$

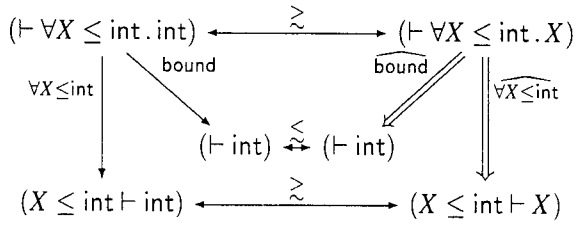
For example, $X \leq \mathsf{int} \vDash \mathsf{int} \gtrsim X$ since:

$$\begin{array}{ccc} (X \leq \mathsf{int} \vdash \mathsf{int}) & \overset{\gtrsim}{\leftarrow} & (X \leq \mathsf{int} \vdash X) \\ {\scriptstyle \mathsf{int}}\downarrow & & \Downarrow{\scriptstyle \widehat{\mathsf{int}}} \\ (X \leq \mathsf{int} \vdash \mathsf{Top}) & \overset{\gtrsim}{\leftarrow} & (X \leq \mathsf{int} \vdash \mathsf{Top}) \end{array}$$

Finally, we are left with the meat of the problem: modelling bounded polymorphism. Modelling Kernel $F_{\mu\leq}$ is not too difficult, we just add transitions which reveal the structure of a polymorphic type:



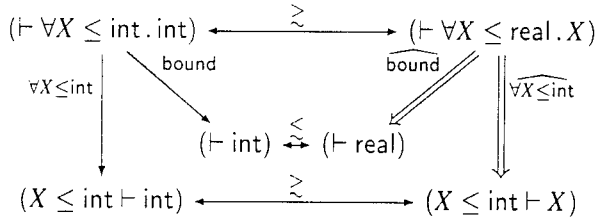
For example, $\models (\forall X \leq \mathsf{int}.\mathsf{int}) \gtrsim (\forall X \leq \mathsf{int}.X)$ since:



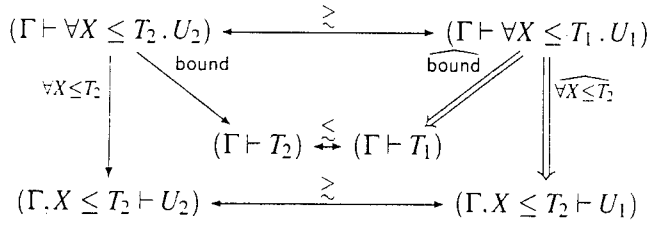
In order to model Full $F_{\mu\leq}$, however, we have to allow the bound of a polymorphic type to vary. We do this by adding an additional transition to the matching transition relation:

$$(\Gamma \vdash \forall X \leq T.U) \stackrel{\widehat{\forall X \leq V}}{\Longrightarrow} (\Gamma, X \leq V \vdash U)$$

For example, $\models (\forall X \leq \mathsf{int}.\mathsf{int}) \gtrsim (\forall X \leq \mathsf{real}.X)$ since:



In general, since bound is a negative label, it is easy to see that the following diagram models the Full $F_{\mu\leq}$ rule for subtyping bounded polymorphism:



As a final example, we consider Ghelli's [9] example of non-termination of the standard algorithm for $F_{\leq}$ subtyping:

$$G \quad = \quad \forall X.\neg(\forall Y \leq X.\neg Y)$$

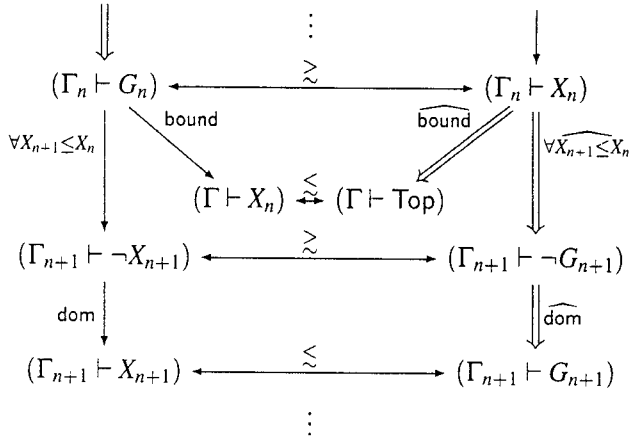
where we write $\neg T$ as shorthand for $T \to \mathsf{Top}$, and $\forall X.T$ as shorthand for $\forall X \leq \mathsf{Top}.T$. Ghelli's example is to verify:

$$X_0 \leq G \models (\forall X_1 \leq X_0.\neg X_1) \gtrsim X_0$$

If we define:

$$\begin{aligned} \Gamma_n &= X_0 \leq G, X_1 \leq X_0, \ldots, X_n \leq X_{n-1} \\ G_n &= \forall X_{n+1} \leq X_n.\neg X_{n+1} \end{aligned}$$

then $\Gamma_n \models G_n \gtrsim X_n$ for every n since:



In particular, $\Gamma_0 \models G_0 \gtrsim X_0$, which is Ghelli's example. Note, however, that in order to show this subtyping, we had to construct an infinite simulation: we cannot just use this lts directly in a model checker to get an algorithm for deciding subtyping of $F_{\mu\leq}$. We will return to this problem in Section 5.

4 Definition of the symbolic lts semantics for $F_{\mu\leq}$

We now provide formal definitions for the material discussed in Section 3. The syntax of positive labels α^+, negative labels α^- and labels α are given:

$$\begin{aligned} \alpha^+ &::= \tau \mid \mathsf{dom} \mid \forall X \leq T \mid X \\ \alpha^- &::= \mathsf{cod} \mid \mathsf{bound} \\ \alpha &::= \alpha^+ \mid \alpha^- \end{aligned}$$

The symbolic lts $\stackrel{\alpha}{\longrightarrow}$ is defined:

$$\begin{aligned} (\Gamma \vdash T \to U) &\stackrel{\mathsf{dom}}{\longrightarrow} (\Gamma \vdash T) \\ (\Gamma \vdash T \to U) &\stackrel{\mathsf{cod}}{\longrightarrow} (\Gamma \vdash U) \\ (\Gamma \vdash K) &\stackrel{K}{\longrightarrow} (\Gamma \vdash \mathsf{Top}) \\ (\Gamma \vdash \forall X \leq T.U) &\stackrel{\mathsf{bound}}{\longrightarrow} (\Gamma \vdash T) \\ (\Gamma \vdash \forall X \leq T.U) &\stackrel{\forall X \leq T}{\longrightarrow} (\Gamma, X \leq T \vdash U) \\ (\Gamma \vdash X) &\stackrel{X}{\longrightarrow} (\Gamma \vdash \mathsf{Top}) \\ (\Gamma \vdash X) &\stackrel{\tau}{\longrightarrow} (\Gamma \vdash \Gamma(X)) \\ (\Gamma \vdash \mu^+ X.T) &\stackrel{\tau}{\longrightarrow} (\Gamma \vdash T[\mu^+ X.T/X]) \end{aligned}$$

The symbolic lts $\xrightarrow{\hat{\alpha}}$ is defined:

$$
\begin{array}{rcl}
(\Gamma \vdash T \to U) & \xrightarrow{\widehat{\mathsf{dom}}} & (\Gamma \vdash T) \\
(\Gamma \vdash T \to U) & \xrightarrow{\widehat{\mathsf{cod}}} & (\Gamma \vdash U) \\
(\Gamma \vdash K) & \xrightarrow{\hat{K'}} & (\Gamma \vdash \mathsf{Top}) \quad (\text{when } K \leq K') \\
(\Gamma \vdash \forall X \leq T.U) & \xrightarrow{\widehat{\mathsf{bound}}} & (\Gamma \vdash T) \\
(\Gamma \vdash \forall X \leq T.U) & \xrightarrow{\widehat{\forall X \leq V}} & (\Gamma, X \leq V \vdash U) \\
(\Gamma \vdash X) & \xrightarrow{\hat{X}} & (\Gamma \vdash \mathsf{Top}) \\
(\Gamma \vdash X) & \xrightarrow{\hat{\tau}} & (\Gamma \vdash \Gamma(X)) \\
(\Gamma \vdash \mu^{+} X.T) & \xrightarrow{\hat{\tau}} & (\Gamma \vdash T[\mu^{+} X.T/X]) \\
(\Gamma \vdash T) & \xrightarrow{\hat{\tau}} & (\Gamma \vdash T)
\end{array}
$$

We write $\Longrightarrow$ for the transitive reflexive closure of $\xrightarrow{\tau}$:

$$\frac{(\Gamma \vdash T) \xrightarrow{\tau} \cdots \xrightarrow{\tau} (\Gamma' \vdash T')}{(\Gamma \vdash T) \Longrightarrow (\Gamma' \vdash T')}$$

We write $\overset{\alpha}{\Longrightarrow}$ for the transition $\xrightarrow{\alpha}$ ignoring τ actions 'on the left', and similarly for $\overset{\hat{\alpha}}{\Longrightarrow}$:

$$\frac{(\Gamma \vdash T) \Longrightarrow \cdot \xrightarrow{\alpha} (\Gamma' \vdash T')}{(\Gamma \vdash T) \overset{\alpha}{\Longrightarrow} (\Gamma' \vdash T')} \qquad \frac{(\Gamma \vdash T) \Longrightarrow \cdot \xrightarrow{\hat{\alpha}} (\Gamma' \vdash T')}{(\Gamma \vdash T) \overset{\hat{\alpha}}{\Longrightarrow} (\Gamma' \vdash T')}$$

A *polar simulation* $\mathcal{R}$ is a well-formed relation on types such that we can complete the diagram:

$$
\begin{array}{ccc}
(\Gamma \vdash T_1) & \overset{\mathcal{R}}{\leftrightarrow} & (\Gamma \vdash T_2) \\
{\scriptstyle \alpha^{\mp}}\downarrow & & \\
(\Gamma' \vdash T_1') & &
\end{array}
\quad \text{as} \quad
\begin{array}{ccc}
(\Gamma \vdash T_1) & \overset{\mathcal{R}}{\leftrightarrow} & (\Gamma \vdash T_2) \\
{\scriptstyle \alpha^{\mp}}\downarrow & & \Downarrow{\scriptstyle \widehat{\alpha^{\mp}}} \\
(\Gamma' \vdash T_1') & \overset{\mathcal{R}^{\mp}}{\leftrightarrow} & (\Gamma' \vdash T_2')
\end{array}
$$

where we write $\mathcal{R}^{\pm}$ for:

$$\frac{(\Gamma \vdash T)\ \mathcal{R}\ (\Gamma \vdash U)}{(\Gamma \vdash T)\ \mathcal{R}^{+}\ (\Gamma \vdash U)} \qquad \frac{(\Gamma \vdash T)\ \mathcal{R}\ (\Gamma \vdash U)}{(\Gamma \vdash U)\ \mathcal{R}^{-}\ (\Gamma \vdash T)}$$

Let $\gtrsim$ be the largest polar simulation.

Proposition 2 *$\gtrsim$ is a preorder.*

Proposition 3 *$\Gamma \vDash T \gtrsim U$ iff $\Gamma \vDash U \sqsubseteq T$.*

5 Motivation for polar simulation up to polarized substitution

We have now given an alternative characterization of coinductive subtyping of $F_{\mu\leq}$, but this does not directly give us any benefits. We can now use standard model-checking techniques to check subtyping, but these only terminate when they find a finite polar simulation. As the Ghelli's example (discussed in Section 3) shows, we can construct types which generate an infinite polar simulation.

In this section, we shall provide a proof technique based on Milner and Sangiorgi's [18] *bisimulation up to* methodology, which can be used to find finite representations of infinite polar simulations. It is based on the requirement to find finite symbolic graphs for process terms in Hennessy and Lin's work [11].

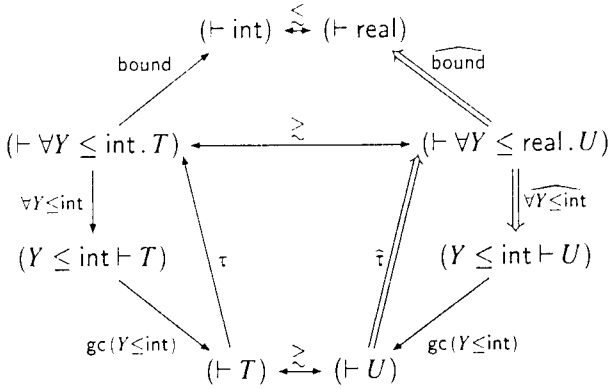
Polar simulation up to garbage collection. Define the *garbage collection* relation on well-formed types as discarding unused type variables, for example:

$$(X \leq \mathsf{int}, Y \leq \mathsf{real} \vdash X) \xrightarrow{\mathsf{gc}(Y \leq \mathsf{real})} (X \leq \mathsf{int} \vdash X)$$

We can use polar simulation up to garbage collection to provide finite proofs of subtyping, for example if we define:

$$T = \mu^{+} X.\forall Y \leq \mathsf{int}.X \qquad U = \mu^{-} X.\forall Y \leq \mathsf{real}.X$$

then we have a finite proof that $\vDash T \gtrsim U$ given by:

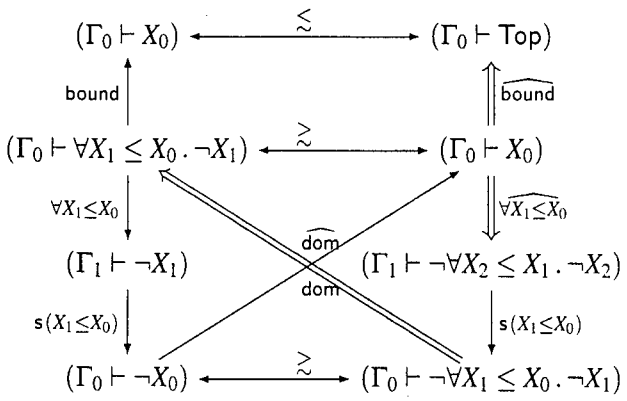


which provides us with a finite representation of the proof that $\vDash T \gtrsim U$. Polar simulation up to garbage collection is a sound proof technique, but it does not cope with Ghelli's example, since there are no unused type variables.

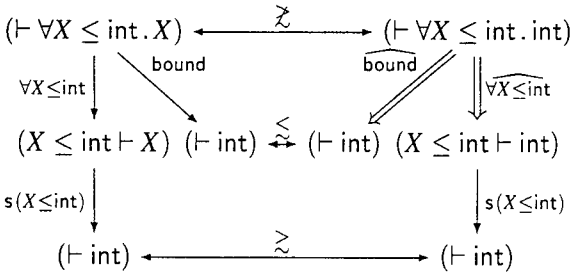
Polar simulation up to substitution. Our next failed attempt to find a proof technique generalizes the notion of polar simulation up to garbage collection, by observing that one can often replace a type variable by its bound, for example:

$$(X \leq \mathsf{int}, Y \leq X \vdash X \to Y) \xrightarrow{\mathsf{s}(X \leq \mathsf{int})} (Y \leq \mathsf{int} \vdash \mathsf{int} \to Y)$$

We can try to use this to show subtypings, for example Ghelli's $\Gamma_0 \vDash G_0 \gtrsim X_0$ from Section 3 has a finite polar simulation up to substitution:



Unfortunately, polar simulation up to substitution is not a sound proof technique, for example:



As this example shows, we cannot always just replace type variables by their bounds, and expect to get a valid subtype relationship.

Polar simulation up to polar substitution. The technique we adopt in this paper is a refinement of polar simulation up to substitution. The crucial observation is that polar simulation up to substitution is sound, as long as we only replace negative occurrences of variables in the supertype, and positive occurrences of variables in the subtype.

Define the *positive substitution* relation as replacing any positive occurrences of a type variable by its bound, and undefined if there are any negative occurrences, for example:

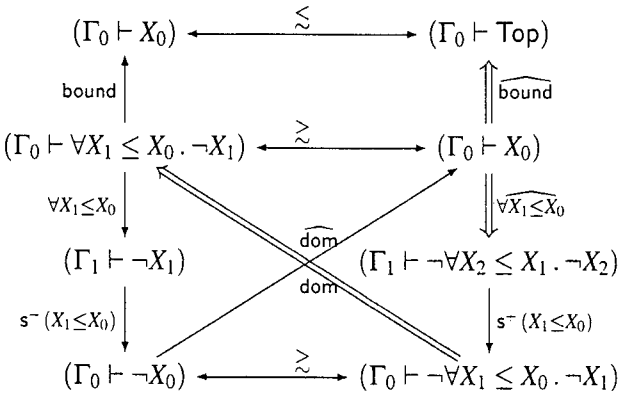
$$(X \le \mathsf{int}, Y \le X \vdash Y \to X) \xrightarrow{s^+(X \le \mathsf{int})} (Y \le \mathsf{int} \vdash Y \to \mathsf{int})$$

$$(X \le \mathsf{int}, Y \le X \vdash X \to Y) \xrightarrow{s^+(X \le \mathsf{int})} \not{} \quad (Y \le \mathsf{int} \vdash \mathsf{int} \to Y)$$

and the *negative substitution* relation similarly (but note that we always substitute positively in the type context):

$$(X \le \mathsf{int}, Y \le X \vdash X \to Y) \xrightarrow{s^-(X \le \mathsf{int})} (Y \le \mathsf{int} \vdash \mathsf{int} \to Y)$$

Then a *polar simulation up to polar substitution* is one where we are allowed to use negative substitution in the supertype, and positive substitution in the subtype. For example, we now have a valid finite proof of Ghelli's example:



and the counterexample for polar simulation up to substitution is no longer a counterexample, because it does not use substitution with the right polarity.

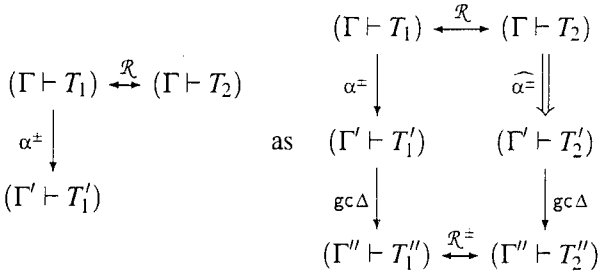
Polar simulation up to polar substitution is the proof technique we adopt for the rest of this paper.

6 Definition of polar simulation up to polar substitution

Let the *garbage collection relation* $(\Gamma \vdash T) \xrightarrow{\mathsf{gc}\Delta} (\Gamma' \vdash T')$ be:

$$(\Gamma, \Delta \vdash T) \xrightarrow{\mathsf{gc}\Delta} (\Gamma \vdash T) \quad (\text{when } \Gamma \vdash T)$$

Let $\mathcal{R}$ be a polar simulation *up to garbage collection* whenever we can complete any diagram:



Define a *polar substitution* $T[U/X]^{\pm}$ as:

$$T[U/X]^{\pm} = T[U/X] \quad (\text{when } X \notin \mathsf{fv}^{\mp}(T))$$

Define a *polar context substitution* $T[\Delta]^{\pm}$ as:

$$T[\emptyset]^{\pm} = T$$
$$T[\Delta, X \le U]^{\pm} = T[U/X]^{\pm}[\Delta]^{\pm} \quad (\text{when } X \notin \mathsf{fv}(\Delta))$$

Define a *polar substitution relation* $(\Gamma \vdash T) \xrightarrow{s^{\pm}\Delta} (\Gamma' \vdash T')$ as:

$$(\Gamma, \Delta \vdash T) \xrightarrow{s^{\pm}\Delta} (\Gamma[\Delta]^{+} \vdash T[\Delta]^{\pm})$$

Note that polar substitution generalizes garbage collection:

$$\text{if } (\Gamma \vdash T) \xrightarrow{\mathsf{gc}\Delta} (\Gamma' \vdash T') \text{ then } (\Gamma \vdash T) \xrightarrow{\mathsf{s}^{\pm}\Delta} (\Gamma' \vdash T')$$

Let $\mathcal{R}$ be a polar simulation *up to polar substitution* whenever we can complete any diagram:

$$\begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\mathcal{R}}{\leftrightarrow} & (\Gamma \vdash T_2) \\ \alpha^{\mp}\downarrow & & \\ (\Gamma' \vdash T_1') & & \end{array} \quad \text{as} \quad \begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\mathcal{R}}{\longleftrightarrow} & (\Gamma \vdash T_2) \\ \alpha^{\mp}\downarrow & & \Downarrow\widehat{\alpha^{\pm}} \\ (\Gamma' \vdash T_1') & & (\Gamma' \vdash T_2') \\ \mathsf{s}^{\mp}\Delta\downarrow & & \downarrow\mathsf{s}^{\pm}\Delta \\ (\Gamma'' \vdash T_1'') & \overset{\mathcal{R}^{\mp}}{\longleftrightarrow} & (\Gamma'' \vdash T_2'') \end{array}$$

We can then show that polar simulation up to polar substitution (and hence up to garbage collection) is a sound proof technique.

Proposition 4 *If $\mathcal{R}$ is a polar simulation up to polar substitution and $\Gamma \vDash T \; \mathcal{R} \; U$ then $\Gamma \vDash T \gtrsim U$.*

7 An algorithm for finding polar simulation up to polar substitution

Polar simulation up to polar substitution gives us a proof technique for showing subtyping, which can easily be converted into a model checking algorithm. Since $F_{\mu\leq}$ is deterministic, a simple breadth-first search algorithm is sufficient. The algorithm is given in Figure 1. The invariants for the **while** loop in the algorithm are:

1. Either $\Gamma_0 \vDash T_0 \; \mathcal{R} \; U_0$ or $\Gamma_0 \vDash T_0 \; \mathcal{S} \; U_0$.
2. $\mathcal{R}$ is a polar simulation up to polar substitution mod $\mathcal{S}$.
3. If $\Gamma_0 \vDash T_0 \gtrsim U_0$ then $(\mathcal{R} \cup \mathcal{S}) \subseteq \gtrsim$.

where $\mathcal{R}$ is a polar simulation up to polar substitution *mod* $\mathcal{S}$ whenever we can complete any diagram:

$$\begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\mathcal{R}}{\leftrightarrow} & (\Gamma \vdash T_2) \\ \alpha^{\mp}\downarrow & & \\ (\Gamma' \vdash T_1') & & \end{array} \quad \text{as} \quad \begin{array}{ccc} (\Gamma \vdash T_1) & \overset{\mathcal{R}}{\longleftrightarrow} & (\Gamma \vdash T_2) \\ \alpha^{\mp}\downarrow & & \Downarrow\widehat{\alpha^{\mp}} \\ (\Gamma' \vdash T_1') & & (\Gamma' \vdash T_2') \\ \mathsf{s}^{\mp}\Delta\downarrow & & \downarrow\mathsf{s}^{\mp}\Delta \\ (\Gamma'' \vdash T_1'') & \overset{\mathcal{R}^{\mp}\cup\mathcal{S}^{\mp}}{\longleftrightarrow} & (\Gamma'' \vdash T_2'') \end{array}$$

It is not too difficult to establish partial correctness of this algorithm, by establishing Invariants 1–3:

function *suptype* (Γ_0, T_0, U_0) {
  let $\mathcal{R} = \emptyset$;
  let $\mathcal{S} = \{\Gamma_0 \vDash T_0 \; \mathcal{S} \; U_0\}$;
  while $(\mathcal{S} \neq \emptyset)$ {
    let $\mathcal{S}' = \emptyset$;
    foreach $(\Gamma_1 \vDash T_1 \; \mathcal{S} \; U_1)$ {
      foreach $(\Gamma_1 \vdash T_1) \xrightarrow{\alpha^{\pm}} (\Gamma_2 \vdash T_2)$ {
        if $(\alpha^{\pm} = \tau)$ {
          add $\Gamma_2 \vDash T_2 \; \mathcal{S}' \; U_1$ **to** $\mathcal{S}'$;
        } **else if** $(\Gamma_1 \vdash U_1) \overset{\widehat{\alpha^{\pm}}}{\Longrightarrow} (\Gamma_2 \vdash U_2)$ {
          let Δ be the largest type context
          such that $(\Gamma_2 \vdash T_2) \xrightarrow{\mathsf{s}^{\mp}\Delta} (\Gamma_3 \vdash T_3)$
          and $(\Gamma_2 \vdash U_2) \xrightarrow{\mathsf{s}^{\pm}\Delta} (\Gamma_3 \vdash U_3)$;
          add $\Gamma_3 \vDash T_3 \; \mathcal{S}'^{\pm} \; U_3$ **to** $\mathcal{S}'^{\pm}$;
        } **else** {
          return *false*;
        }
      }
    }
    $\mathcal{R} = \mathcal{R} \cup \mathcal{S}$;
    $\mathcal{S} = \mathcal{S}' \setminus \mathcal{R}$;
  }
  return *true*;
}

Figure 1: The algorithm

Proposition 5 *For any $\Gamma_0 \vdash T_0$ and $\Gamma_0 \vdash U_0$ we have:*

1. *If suptype (Γ_0, T_0, U_0) returns true then $\Gamma_0 \vDash T_0 \gtrsim U_0$.*
2. *If suptype (Γ_0, T_0, U_0) returns false then $\Gamma_0 \vDash T_0 \not\gtrsim U_0$.*

We can show that the algorithm is guaranteed to terminate in the case where $\Gamma \vDash T \not\gtrsim U$.

Proposition 6 *If $\Gamma \vDash T \not\gtrsim U$ then suptype (Γ, T, U) terminates.*

We can also show that if there is a finite polar simulation up to polar substitution, then the algorithm will find it, and so will terminate. For example, this means the algorithm is guaranteed to terminate on Ghelli's example.

Proposition 7 *If there exists a finite polar simulation up to polar substitution $\mathcal{R}_f$ such that $\Gamma \vDash T \; \mathcal{R}_f \; U$ then suptype (Γ, T, U) terminates.*

Using this, we can show that the algorithm is at least as strong as the standard algorithm for subtyping $F_{\leq}$. We do this by showing that if $\Gamma \vdash T \geq U$ then we can construct a finite polar simulation $\mathcal{R}$ such that $\Gamma \vdash T \mathcal{R} U$.

Proposition 8 *If the standard algorithm for subtyping $F_{\leq}$ terminates, then suptype (Γ, T, U) terminates with the same result.*

Since our algorithm is at least as powerful as the standard algorithm, but terminates on Ghelli's example, we have that our example is strictly more powerful.

8 Kernel $F_{\mu\leq}$

In [4], Colazzo and Ghelli provide an algorithm for subtyping of Kernel $F_{\mu\leq}$. Their algorithm:

- Works directly on the structure of the types, rather than via an lts semantics.
- Does not work 'up to α-conversion', which results in a more efficient algorithm, at the cost of extra complexity.

We can easily modify our algorithm to check Kernel $F_{\mu\leq}$ subtyping, by changing the matching transition rule for polymorphic types to require bounds to be matched exactly:

$$(\Gamma \vdash \forall X \leq T . U) \xrightarrow{\widehat{\forall X \leq T}} (\Gamma, X \leq T \vdash U)$$

We can show that this modified algorithm is as powerful as theirs (although probably not as efficient, depending on how α-conversion is handled), by showing that our algorithm terminates on Kernel $F_{\mu\leq}$.

Proposition 9 *If $\Gamma \vDash T \gtrsim U$ in Kernel $F_{\mu\leq}$, then there is a finite polar simulation $\mathcal{R}$ up to garbage collection such that $\Gamma \vDash T \mathcal{R} U$.*

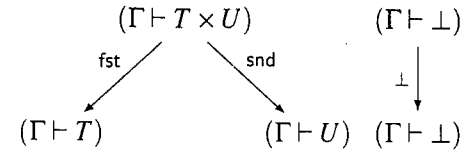
Together with Proposition 7, this gives us that our algorithm is a decision procedure for subtyping of Kernel $F_{\mu\leq}$.

Proposition 10 *If $\Gamma \vDash T \gtrsim U$ in Kernel $F_{\mu\leq}$, then suptype (Γ, T, U) terminates with true.*

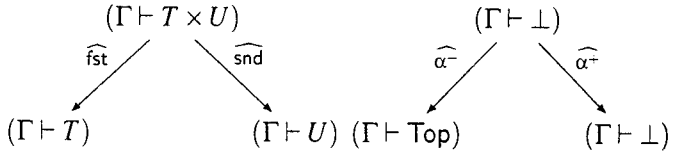
9 Colazzo and Ghelli's benchmark examples

We have already shown that our algorithm terminates on Ghelli's example of nontermination of the standard subtyping algorithm for $F_{\leq}$.

Colazzo and Ghelli [4] provide two motivating examples for their algorithm for Kernel $F_{\mu\leq}$, which act as useful benchmarks for our approach. The examples make use of tuple types $T \times U$, and a bottom type $\bot$: these can easily be given an lts semantics:



with matching transitions:



For example, we can use this semantics to verify one of Pierce's [15] requirements for subtyping with $\bot$, that any type variable bounded by $\bot$ is equivalent to $\bot$:

$$X \leq \bot \vDash X \gtrsim \bot \qquad X \leq \bot \vDash \bot \gtrsim X$$

In the examples, we also use many syntactic abbreviations, such as defining equations, missing Top bounds, and ignoring some τ steps.

The first example is a benchmark which checks that the algorithm performs enough garbage collection to find a finite polar simulation up to garbage collection. It is given in Figure 2.

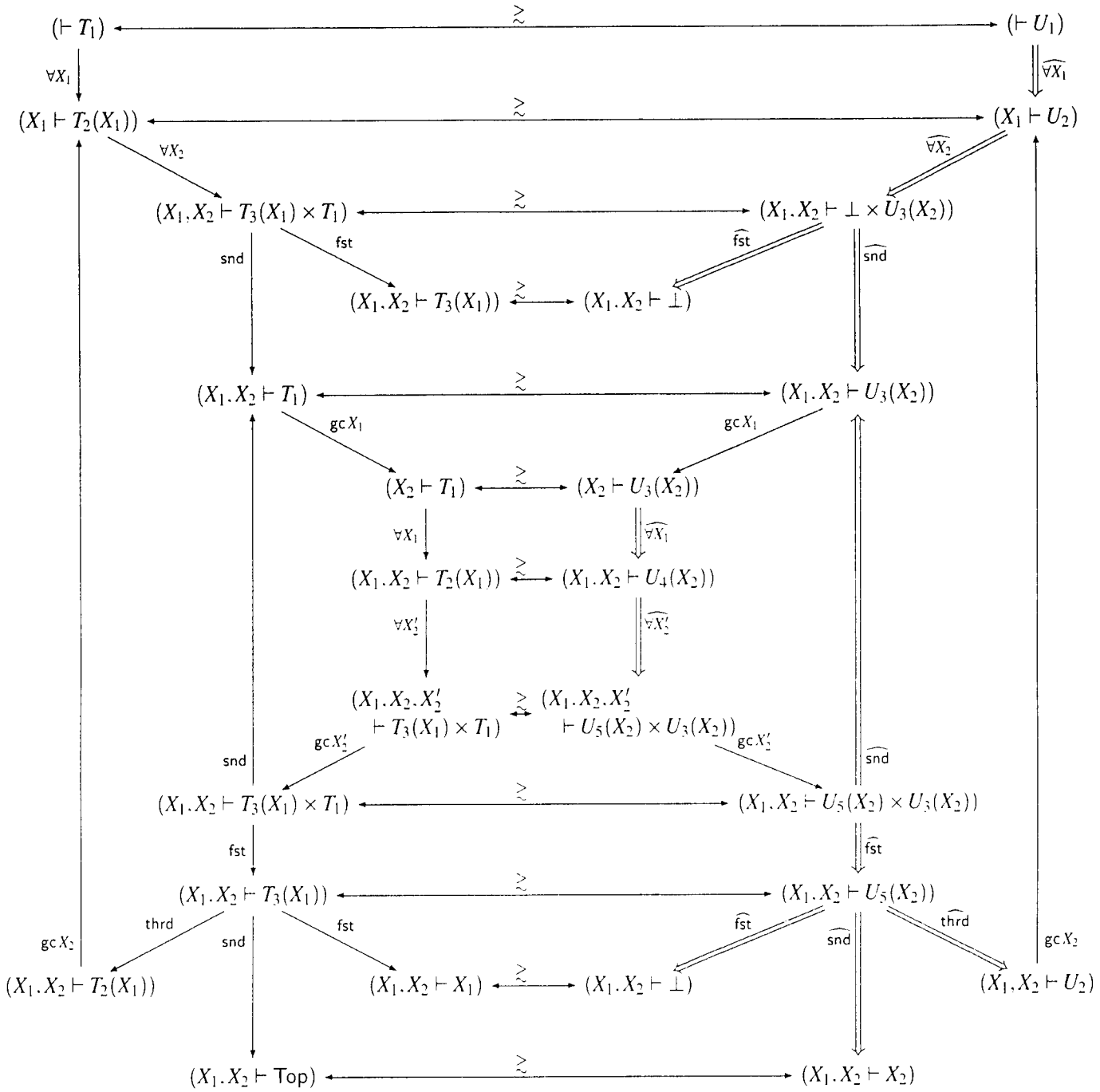
The second example checks that the algorithm does not produce false positives, caused by collapsing variables together incorrectly. It is given in Figure 3.

10 Conclusions and further work

This paper describes an application of symbolic labelled transition systems, which have previously been used to model concurrent languages, to modelling subtyping. This allows us to use the techniques from concurrency theory, such as simulations, and 'simulation up to' to reason about subtyping. It also often makes proofs easier to read, even in the presence of quite complex types such as Colazzo and Ghelli's benchmark in Figure 2.

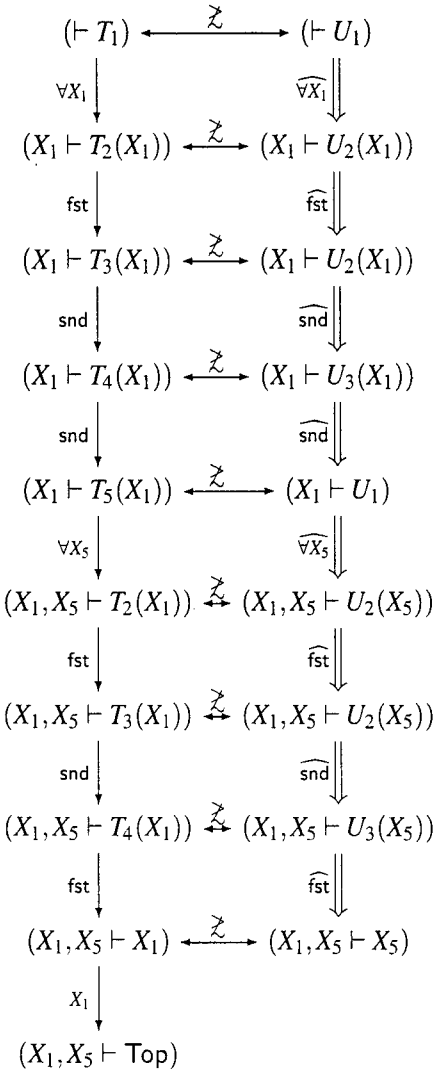
This technique should generalize to other examples such as record subtyping, union types and intersection types. It may be that Gordon's [10] work on lts semantics for λ-calculi could be applied here, to give a semantics of higher-order features such as functions of kind Type $\to$ Type. We leave the technical development of this to future work.

The main result which is missing from the current work is a syntactic characterization of when the algorithm *suptype* terminates. Also, we have not discussed how α-conversion would be implemented: it should be possible to define α-conversion as a strong bisimulation, and then use polar simulation up to strong bisimulation as a proof technique. We also leave these issues for future work.



$$
\begin{array}{rcl@{\qquad}rcl}
T_1 & \stackrel{\text{def}}{=} & \forall X_1 \,.\, T_2(X_1) & U_1 & \stackrel{\text{def}}{=} & \forall Y_1 \,.\, U_2 \\
T_2(X_1) & \stackrel{\text{def}}{=} & \forall X_2 \,.\, (T_3(X_1) \times T_1) & U_2 & \stackrel{\text{def}}{=} & \forall Y_2 \,.\, (\bot \times U_3(Y_2)) \\
T_3(X_1) & \stackrel{\text{def}}{=} & X_1 \times \mathsf{Top} \times T_2(X_1) & U_3(Y_2) & \stackrel{\text{def}}{=} & \forall Y_3 \,.\, U_4(Y_2) \\
 & & & U_4(Y_2) & \stackrel{\text{def}}{=} & \forall Y_4 \,.\, (U_5(Y_2) \times U_3(Y_2)) \\
 & & & U_5(Y_2) & \stackrel{\text{def}}{=} & \bot \times Y_2 \times U_2
\end{array}
$$

Figure 2: Colazzo and Ghelli's first example: show that $\vDash T_1 \gtrsim U_1$



$$
\begin{aligned}
T_1 &\stackrel{\text{def}}{=} \forall X_1 . T_2(X_1) \\
T_2(X_1) &\stackrel{\text{def}}{=} T_3(X_1) \times \text{Top} \\
T_3(X_1) &\stackrel{\text{def}}{=} \text{Top} \times T_4(X_1) \\
T_4(X_1) &\stackrel{\text{def}}{=} X_1 \times T_5(X_1) \\
T_5(X_1) &\stackrel{\text{def}}{=} \forall X_5 . T_2(X_1)
\end{aligned}
$$

$$
\begin{aligned}
U_1 &\stackrel{\text{def}}{=} \forall Y_1 . U_2(Y_1) \\
U_2(Y_1) &\stackrel{\text{def}}{=} U_2(Y_1) \times U_3(Y_1) \\
U_3(Y_1) &\stackrel{\text{def}}{=} Y_1 \times U_1
\end{aligned}
$$

Figure 3: Colazzo and Ghelli's second example: show $\vDash T_1 \not\sim U_1$

References

[1] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subbtyping. In *Proc. Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag, 1997.

[2] K. B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Inform. and Comput.*, 87(1):196–240, 1990.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[4] D. Colazzo and G. Ghelli. Subtyping recursive types in kernel Fun, extended abstract. In *Proc. Logic in Computer Science*. IEEE Computer Society Press, 1999.

[5] P.-L. Curien and G. Ghelli. Coherence of subsumtion: Minimum typing and type checking in $F_{\leq}$. *Math. Struct. in Comp. Sci.*, 2(1):55–91, 1992.

[6] V. Gapeyev, M. Levin, and B. C. Pierce. Recursive subtyping revealed. In *Proc. Int. Conf. Functional Programming*, 2000.

[7] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Universitá di Pisa, 1990.

[8] G. Ghelli. Recursive types are not conservative over $F_{\leq}$. In M. Bezen and J.F. Groote, editors, *Proc. Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 146–162. Springer-Verlag, 1993.

[9] G. Ghelli. Divergence of $\leq$ type checking. *Theoret. Comput. Sci.*, 139(1-2):131–162, 1995.

[10] A. D. Gordon. Bisimilarity as a theory of functional programming. In *Proc. Math. Foundations of Programming Semantics*, number 1 in Electronic Notes in Comp. Sci. Elsevier, 1995.

[11] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoret. Comput. Sci.*, pages 353–389, 1995.

[12] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[13] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[14] B. C. Pierce. Bounded quantification is undecidable. *Inform. and Comput.*, 112(1):131–165, 1994.

[15] B. C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

[16] B.C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proc. LICS '93*, pages 376–385. IEEE Computer Society Press, 1993.

[17] M. Amadio R and L. Cardelli. Subtyping recursive types. *ACM Trans. Programming Languages and Systems*, 15(4):575–631, 1993.

[18] D. Sangiorgi and R. Milner. The problem of 'weak bisimulation up to'. In *Proc. CONCUR 92*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[19] P. Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, Computer Laboratory, University of Cambridge, 1997.

[20] D. N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

A continuum of theories of lambda calculus without semantics

Antonino Salibra

Dipartimento di Informatica, Università di Venezia
Via Torino 155, 30172 Venezia, Italy
salibra@dsi.unive.it

Abstract

In this paper we give a topological proof of the following result: There exist $2^{\aleph_0}$ lambda theories of the untyped lambda calculus without a model in any semantics based on Scott's view of models as partially ordered sets and of functions as monotonic functions. As a consequence of this result, we positively solve the conjecture, stated by Bastonero-Gouy [6, 7] and by Berline [10], that the strongly stable semantics is incomplete.

1. Introduction

Lambda theories are consistent extensions of the lambda calculus that are closed under derivation. They arise by syntactical considerations, a lambda theory may correspond to a possible operational (observational) semantics of lambda calculus (see e.g. [2, 3, 24]), as well as by semantic ones, a lambda theory may be the theory of a model of lambda calculus (see e.g. [3, 10]). Since the lattice of lambda theories is a very rich and complex structure (see e.g. [3, 10, 24, 25, 49]), syntactical techniques are usually difficult to use in the study of lambda theories. Therefore, semantic methods have been extensively investigated.

Computational motivations and intuitions justify Scott's view of models (see [44, 45]) as partially ordered sets (sets of observations or informations) and of computable functions as monotonic functions over these sets. After Scott, mathematical models of lambda calculus in various categories of domains (see [1, 48]) were classified into semantics according to the nature of their representable functions (see [2, 3, 4, 10, 16, 20, 25]). Scott's continuous semantics [45] is given in the category whose objects are complete partial orders and morphisms are continuous functions. The stable semantics introduced by Berry in [11] and the recent strongly stable semantics introduced by Bucciarelli and Ehrhard in [12] are strengthening of the continuous semantics. The stable semantics is given in the category of DI-domains with stable functions as morphisms, while the strongly stable one in the category of DI-domains with coherence, and strongly stable functions as morphisms. All these semantics are structurally and equationally rich in the sense that it is possible to build up $2^{\aleph_0}$ models in each of them inducing pairwise distinct lambda theories (see [28, 29]). The problem of the equational richness is related to the problem of the completeness/incompleteness of a semantics: are the set of lambda theories determined by these semantics equal or strictly included within the set of consistent lambda theories?

The first incompleteness result was obtained by Honsell and Ronchi della Rocca [25] for the continuous semantics. They proved, via a hard syntactical proof, that the contextual lambda theory induced by the set of essentially closed terms does not admit a continuous model. Following a similar method, Gouy [21] proved the incompleteness of the stable semantics with a much harder syntactical proof. Other more semantic proofs of incompleteness for the continuous and stable semantics can be found in [7]. Bastonero [6] provides an incompleteness result for the hypercoherence semantics.

Bastonero [6, Section 6], Bastonero-Gouy [7, Section 7] and Berline [10, Section 6.1] conjecture that the strongly stable semantics is also incomplete. In this paper we give a positive answer to this open question. We prove that any semantics of lambda calculus based on Scott's paradigmatic view of models as partially ordered sets and of computable functions as monotonic functions is incomplete if the partial order admits a bottom element. This incompleteness is due to $2^{\aleph_0}$ distinct lambda theories. The main theorem of the paper unifies and subsumes incompleteness results for different classes of models that have been proved in different ways, using different approaches.

The proof of incompleteness is based on a general theorem of separation for topological algebras. We prove that under a very weak condition, called weak subtractivity, a topological algebra admits two elements 0 and 1 which can be $T_{2_{1/2}}$-separated (i.e., there exist two open neighbourhoods of 0 and 1 respectively whose closures have empty intersection). All models of lambda calculus based on Scott's paradigmatic view are topological algebras with respect to the Alexandroff topology generated

0-7695-1281-X/01 $10.00 © 2001 IEEE

by the partial order over the model. Posets such as join semilattices, meet semilattices, complete partial orderings, lattices, posets with a least element, posets with a greatest element cannot have $T_{2_{1/2}}$-separated elements w.r.t. the Alexandroff topology. Then the incompleteness theorem is determined by proving that there exist $2^{\aleph_0}$ semisensible lambda theories that admit only weakly subtractive models.

2. Preliminaries

To keep this article self-contained, we summarize some definitions and results that we will need in the subsequent part of the paper. With regard to the lambda calculus we follow the notation and terminology of Barendregt (see [3]).

For the general theory of lambda calculus the reader may consult Barendregt [3] and Krivine [30]. For the general theory of universal algebras the reader may consult Burris and Sankappanavar [13], Gratzer [22], and McKenzie, McNulty and Taylor [32]. The main references for topological algebras are Taylor [52, 53], Gumm [23], Bentz [8] and Coleman [14, 15].

2.1. Lambda theories

Λ denotes the set of λ-terms, while Λ^o denotes the set of closed λ-terms, where a λ-term is closed if it does not admit free occurrences of variables.

Lambda theories are consistent extensions of the lambda calculus that are closed under derivation. Remember that an equation is a formula of the form $M = N$ with $M, N \in \Lambda$. The equation is closed if M and N are closed λ-terms. If $\mathcal{T}$ is a set of equations, then the theory $\lambda + \mathcal{T}$ is obtained by adding to the axioms and rules of the lambda calculus the equations in $\mathcal{T}$ as new axioms. If $\mathcal{T}$ is a set of closed equations, $\mathcal{T}^+$ is the set of closed equations provable in $\lambda + \mathcal{T}$. $\mathcal{T}$ is a lambda theory if $\mathcal{T}^+ = \mathcal{T}$ (see [3, Def. 4.1.1]). As a matter of notation, $\mathcal{T} \vdash M = N$ stands for $\lambda + \mathcal{T} \vdash M = N$; this is also written as $M =_{\mathcal{T}} N$. $[M]^{\mathcal{T}}_{\mathcal{T}} = \{N \in \Lambda^o : \mathcal{T} \vdash N = M\}$ denotes the equivalence class of the closed λ-term M.

The lambda theory $\mathcal{H}$, generated by equating all the unsolvable λ-terms, is consistent [3, Thm. 16.1.3]. A lambda theory $\mathcal{T}$ is called semisensible [3, Def. 4.1.7(iii)] if $\mathcal{T} \nvdash M = N$ whenever M is solvable and N is unsolvable.

2.2. Combinatory algebras and λ-models

An algebra $\mathbf{C} = (C, \cdot, \mathbf{k}, \mathbf{s})$, where $\cdot$ is a binary operation and $\mathbf{k}, \mathbf{s}$ are constants, is called a *combinatory algebra* (Curry [17], Schönfinkel [43]) if it satisfies the following identities (as usual the symbol $\cdot$ is omitted, and association is to the left): $\mathbf{k}xy = x; \mathbf{s}xyz = xz(yz)$. In the equational language of combinatory algebras the derived combinator $\mathbf{1}$ is defined as $\mathbf{1} \equiv \mathbf{s}(\mathbf{ki})$. A function $f : C \to C$ is called *representable* if there exists an element $c \in C$ such that $cz = f(z)$ for all $z \in C$. If this last condition is satisfied, we say that c represents map f in $\mathbf{C}$.

Let $\mathbf{C}$ be a combinatory algebra and let $\bar{c}$ be a new symbol for each $c \in C$. Extend the language of lambda calculus by adjoining $\bar{c}$ as a new constant symbol for each $c \in C$. Let $\Lambda^o(C)$ be the set of closed λ-terms with constants from C. The interpretation of terms in $\Lambda^o(C)$ with elements of C can be defined by induction as follows (for all $M, N \in \Lambda^o(C)$ and $c \in C$):

$$|\bar{c}|_{\mathbf{C}} = c;\ |(MN)|_{\mathbf{C}} = |M|_{\mathbf{C}}|N|_{\mathbf{C}};\ |\lambda x.M|_{\mathbf{C}} = \mathbf{1}m,$$

where $m \in C$ is any element representing the following map $f : C \to C$:

$$f(c) = |M[x := \bar{c}]|_{\mathbf{C}}, \quad \text{for all } c \in C.$$

The drawback of the previous definition is that, if $\mathbf{C}$ is an arbitrary combinatory algebra, it may happen that map f is not representable. The axioms of a subclass of combinatory algebras, called *λ-models* or models of lambda calculus (Meyer [33], Scott [47], [3, Def. 5.2.7]), were expressly chosen to make coherent the previous definition of interpretation. For every λ-model $\mathbf{C}$, the set $Th(\mathbf{C}) = \{M = N : M, N \in \Lambda^o,\ \mathbf{C} \models M = N\}$ constitutes a lambda theory. $\mathbf{C}$ is a model of the lambda theory $\mathcal{T}$ if $\mathcal{T} = Th(\mathbf{C})$.

We would like to point out here that there exists an algebraic approach to the model theory of lambda calculus, alternative to combinatory logic, that allows to keep the lambda notation and all the functional intuitions (see [34, 35, 36, 40, 41, 42]).

2.3. Topological algebras

A topological algebra is a pair $(\mathbf{A}, \tau)$ where $\mathbf{A}$ is an algebra and τ is a topology on the underlying set A with the property that each basic operation of $\mathbf{A}$ is continuous with respect to τ. (We will occasionally avoid explicit mention of τ.)

Let $\bar{b}$ be the closure of set $\{b\}$. For any (topological) space (A, τ) a preorder can be defined by

$$a \leq_\tau b \text{ iff } a \in \bar{b} \text{ iff } \forall U \in \tau (a \in U \Rightarrow b \in U).$$

We have

$$\tau \text{ is } T_0 \text{ iff } \leq_\tau \text{ is a partial order.}$$

For any T_0-space A the partial order $\leq_\tau$ is called *the specialization order* of τ. Note that any continuous map between T_0-spaces is necessarily monotone and that the order is discrete (i.e. satisfies $a \leq_\tau b$ iff $a = b$) iff A is a T_1-space.

Let $(A, \leq)$ be a partially ordered set (poset). $B \subseteq A$ is an upper (lower) set if $b \in B$ and $b \leq a$ ($a \leq b$) imply

$a \in B$. We utilize the notation $B\uparrow$ ($B\downarrow$) for the least upper (lower) set containing a subset B of A. We write $a\uparrow$ for $\{a\}\uparrow$ and $a\downarrow$ for $\{a\}\downarrow$.

Given a poset $(A, \leq)$ we can find many T_0-topologies τ on A for which $\leq$ is the specialization ordering of τ (see Johnstone [27, Section II.1.8]). The Alexandroff topology and the weak topology defined below are the maximal one and the minimal one with this property.

The Alexandroff topology $\tau_{(A,\leq)}$ is constituted by the collection of all upper sets in A, i.e.,

$$U \text{ is an Alexandroff open iff } U = U\uparrow.$$

Then $a\uparrow$ is the least open set containing a. A function is continuous w.r.t. the Alexandroff topology if, and only if, it is monotonic. The closure of the open set $a\uparrow$ is $(a\uparrow)\downarrow$.

The weak topology $w_{(A,\leq)}$ is constituted by the smallest topology for which all sets of the form $a\downarrow$ are closed, i.e. the topology based by sets of the form $A-(a_1\downarrow\cup\cdots\cup a_k\downarrow)$.

Let $(A, \leq)$ be a poset, τ be a topology on A. Then τ is T_0 with specialization order $\leq$ if, and only if, $w_{(A,\leq)} \subseteq \tau \subseteq \tau_{(A,\leq)}$.

3. The topological theorem

In this Section we prove a general theorem of separation for topological algebras. Under a very weak condition, called *weak subtractivity*, a topological algebra admits two elements 0 and 1 which can be $T_{2_{1/2}}$-separated. We were inspired with Bentz [8] and Coleman [14, 15] for the idea of this theorem and for the techniques used in its proof. In the last part of the Section we characterize the topological algebras with Alexandroff topology which cannot be weakly subtractive.

The notion of subtractivity in Universal Algebra was introduced by Aldo Ursini [54]. A variety (equational class) of algebras is *subtractive* if there exist a term $s(x, y)$ and a constant 0 such that the identities

$$s(x, x) = 0; \quad s(x, 0) = x$$

are satisfied by every algebra in the variety. Term s simulates part of subtraction: x minus x is equal to 0, while x minus 0 is equal to x.

In this paper we introduce a weak form of subtractivity.

Definition 3.1 *An algebra* $\mathbf{A}$ *is weakly subtractive if there exist a term* $s(x, y)$ *and two constants* 0 *and* 1 *in the similarity type of* $\mathbf{A}$ *such that*

$$s(x, x) = 0; \quad s(1, 0) = 1; \quad 1 \neq 0.$$

Separation axioms in topology stipulate the degree to which distinct points may be separated by open sets or by closed neighborhoods of open sets. In the following theorem we prove that in every weakly subtractive T_0-topological algebra the elements 0 and 1 can be $T_{2_{1/2}}$-separated. This means that there exist two open neighbourhoods of 0 and 1 respectively whose closures have empty intersection.

As a matter of notation, if A is a space then the closure of a subset U of A will be denoted by $\overline{U}$. Recall that $a \in \overline{U}$ if $U \cap V \neq \emptyset$ for every open neighbourhood V of a.

Theorem 3.1 *Let* $(\mathbf{A}, \tau)$ *be a weakly subtractive* T_0*-topological algebra. Then there exist an open neighbourhood* V *of* 1 *and an open neighbourhood* W *of* 0 *such that* $\overline{V} \cap \overline{W} = \emptyset$.

Proof: The proof is divided into claims.

Claim 3.1 *There exists an open neighbourhood* U *of* 1 *such that* $0 \notin U$.

Assume, by the way of contradiction, that $1 \leq_\tau 0$, i.e., every open neighbourhood of 1 contains 0. Then by the T_0 hypothesis on τ there exists an open neighbourhood Z of 0 such that $1 \notin Z$. Then we have $0 = s(1, 1) \in Z$. By continuity in the second coordinate, there exists an open neighbourhood R of 1 such that $s(1, R) \subseteq Z$. By $1 \leq_\tau 0$ it follows that $0 \in R$, so that $1 = s(1, 0) \in Z$. Contradiction.

Claim 3.2 *There exist an open neighbourhood* V' *of* 1 *and an open neighbourhood* W' *of* 0 *such that* $V' \cap W' = \emptyset$.

By Claim 3.1 there exists an open neighbourhood U of 1 such that $0 \notin U$. From $s(1, 0) = 1 \in U$ and from the continuity of s it follows that there exist two open neighbourhoods V', W' of 1 and 0 respectively such that $s(V', W') \subseteq U$. If there is an element $b \in V' \cap W'$ then $0 = s(b, b) \in U$ that contradicts the hypothesis on U. Then we have $V' \cap W' = \emptyset$.

We now provide the proof of the theorem. By Claim 3.2 there exist two open neighbourhoods V' and W' of 1 and 0 respectively with empty intersection. Since s is continuous and $s(1, 0) = 1 \in V'$, there exist two other open sets V and W containing 1 and 0, respectively, such that $s(V, W) \subseteq V'$. The sets V and W will be the right sets for the conclusion of the theorem. Since s is continuous the pre-image of $\overline{V'}$ under the map s is closed. From $s(V, W) \subseteq V' \subseteq \overline{V'}$ the pre-image of $\overline{V'}$, that is closed, contains $V \times W$, so $s(\overline{V}, \overline{W}) \subseteq \overline{V'}$.

We now prove that $\overline{V} \cap \overline{W} = \emptyset$. Assume, by the way of contradiction, that there is $d \in \overline{V} \cap \overline{W}$. Since $s(\overline{V}, \overline{W}) \subseteq \overline{V'}$ it follows that $0 = s(d, d) \in \overline{V'}$. But by definition of closure of a set this is possible only if for every open neighbourhood Z of 0, we have that $Z \cap V' \neq \emptyset$. But this contradicts our initial choice of V' and W' as two open neighbourhoods of 1 and 0 respectively with empty intersection. □

Connectedness axioms in topology examine the structure of topological spaces in an orthogonal way with respect to separation axioms. They deny the existence of certain subsets of a topological space with properties of separation. For example, a space with no disjoint open sets is called hyperconnected, while a space with no disjoint closed sets is called ultraconnected (see Steen-Seebach [51, Section 4]).

Definition 3.2 *We say that a space is closed-open-connected, co-connected for short, if it has no disjoint closures of open sets. In other words, if, for all open sets U and V, we have that $\overline{V} \cap \overline{U} \neq \emptyset$.*

We have the following implications:

hyperconnectedness $\Rightarrow$ co-connectedness $\Rightarrow$ connectedness

and

ultraconnectedness $\Rightarrow$ co-connectedness $\Rightarrow$ connectedness.

Then co-connectedness is a sort of meeting point between ultraconnectedness and hyperconnectedness.

The following result is an easy consequence of Thm. 3.1.

Corollary 3.1 *There exists no weakly subtractive T_0-topological algebra $(\mathbf{A}, \tau)$ whose topology τ is co-connected.*

We say that a poset $(A, \leq)$ is *co-connected* if the Alexandroff topology $\tau_{(A,\leq)}$ is co-connected. This is equivalent to say that, for all $a, b \in A$, $(a\uparrow)\downarrow \cap (b\uparrow)\downarrow \neq \emptyset$. The following posets are co-connected: join semilattices, meet semilattices, complete partial orderings, lattices, posets with a least element, posets with a greatest element.

By definition a topology τ_1 is weaker than a topology τ_2 if $\tau_1 \subseteq \tau_2$.

Lemma 3.1 *If the topology τ_1 is weaker than a co-connected topology τ_2, then τ_1 is also co-connected.*

Proof: The closure of a set grows up if there are less open (and closed) sets. □

Theorem 3.2 *There exists no weakly subtractive T_0-topological algebra whose specialization order is co-connected.*

Proof: Let $(\mathbf{A}, \tau)$ be a weakly subtractive T_0-topological algebra whose specialization order $\leq$ is co-connected. By Thm. 3.1 there exist an open neighbourhood V of 1 and an open neighbourhood W of 0 such that $\overline{V} \cap \overline{W} = \emptyset$. Then the topology τ is not co-connected. The Alexandroff topology $\tau_{(A,\leq)}$ is the maximal topology τ_1 with the property that $\leq$ is the specialization ordering of τ_1 (see Johnstone [27, Section II.1.8]). Then τ is weaker than the Alexandroff topology $\tau_{(A,\leq)}$. By hypothesis the Alexandroff topology is co-connected. By applying Lemma 3.1 we get that τ is also co-connected. This is a contradiction. □

4. The incompleteness theorem

A class $\mathcal{C}$ of models of lambda calculus *represents a lambda theory* $\mathcal{T}$ if there is a model in $\mathcal{C}$ whose theory is exactly $\mathcal{T}$. A class of models is *incomplete* if it does not represent all the lambda theories.

We now define a class of $2^{\aleph_0}$ semisensible distinct lambda theories satisfying the following condition: if $\mathbf{C}$ is model of a lambda theory in the class, then $\mathbf{C}$ is a weakly subtractive combinatory algebra.

Consider the (consistent and) semisensible lambda theory Δ axiomatized by

$$\Omega xx = \Omega; \quad \Omega\Omega_3\Omega = \Omega_3,$$

where $\Omega \equiv (\lambda x.xx)(\lambda x.xx), \Omega_3 \equiv (\lambda x.xxx)(\lambda x.xxx)$.

In the next theorem, the technically hardest part of the work, we prove that the lambda theory Δ does not equate Ω and Ω_3. This result implies that the term model of Δ is a weakly subtractive combinatory algebra.

Theorem 4.1

$$\Delta \not\vdash \Omega = \Omega_3.$$

Proof: We provide an outline of the proof. Define

$$\Omega_3^* = \Omega\Omega_3\Omega; \quad (\Omega\Omega_3\Omega)^* = \Omega_3. \tag{1}$$

The definition of a context, i.e., a lambda term with some holes in it, can be found in [3, Def. 2.1.18]. Let Σ be the least lambda theory satisfying the following conditions for every context $C[\,]$, λ-term N, and element $d \in \{\Omega_3, \Omega\Omega_3\Omega\}$:

(i) $\Sigma \vdash \Omega xx = \Omega$;

(ii) $\Sigma \vdash \Omega(C[d])N = \Omega$ implies $\Sigma \vdash \Omega(C[d^*])N = \Omega$;

(iii) $\Sigma \vdash \Omega N(C[d]) = \Omega$ implies $\Sigma \vdash \Omega N(C[d^*]) = \Omega$.

Σ exists because the set of lambda theories satisfying the three above conditions is closed under arbitrary intersection and it is not empty (the lambda theory $\mathcal{H}$ equating all the unsolvable satisfies (i)-(iii)).

Σ satisfies the following condition for all λ-terms M, N:

$$\Sigma \vdash M = N \;\Rightarrow\; \Sigma \vdash \Omega MN = \Omega. \tag{2}$$

From $\Sigma \vdash M = N$ and $\Sigma \vdash \Omega NN = \Omega$ it follows that $\Sigma \vdash \Omega MN = \Omega NN = \Omega$.
Let $\rightarrow_\Sigma$ be the following reduction rule:

$$\Omega MN \rightarrow_\Sigma \Omega \tag{3}$$

for every M and N such that $\Sigma \vdash \Omega MN = \Omega$. The reflexive closure of $\rightarrow_\Sigma$ satisfies the diamond property, and the relations $\rightarrow_\beta$ and $\rightarrow_\Sigma$ commute. Then the reduction rule $\rightarrow_{\beta\Sigma} = \rightarrow_\beta \cup \rightarrow_\Sigma$ is Church-Rosser by the Hindley-Rosen Lemma (see Berarducci-Intrigila [9, Thm. 3.4] and Barendregt [3, Prop. 3.3.5]).

Then we prove that Σ is the lambda theory generated by conversion $\cong_{\beta\Sigma}$ from $\rightarrow_{\beta\Sigma}$, i.e.,

$$\Sigma \vdash M = N \text{ iff } M \cong_{\beta\Sigma} N. \tag{4}$$

The proof of (4) is obtained as follows. Since $\Omega MN \rightarrow_\Sigma \Omega$ iff $\Sigma \vdash \Omega MN = \Omega$, then it is obvious that $M \cong_{\beta\Sigma} N$ implies $\Sigma \vdash M = N$. For the opposite direction, we utilize conditions (ii)-(iii) in the definition of Σ to prove that, for every $d \in \{\Omega_3, \Omega\Omega_3\Omega\}$ and every λ-term P,

$$Pd \cong_{\beta\Sigma} \Omega \;\Rightarrow\; Pd^* \cong_{\beta\Sigma} \Omega. \tag{5}$$

Then we use (5) to show that the conversion relation $\cong_{\beta\Sigma}$ satisfies conditions (i)-(iii) utilized in the definition of Σ. Since Σ is the least lambda theory satisfying conditions (i)-(iii) we have the conclusion.

From (4) it follows that

$$\Sigma \not\vdash \Omega = \Omega_3 \tag{6}$$

since Ω and Ω_3 do not have a common reduct w.r.t. $\rightarrow_{\beta\Sigma}$. The next step in the proof is to show that

$$\Sigma + \Omega\Omega_3\Omega = \Omega_3 \not\vdash \Omega = \Omega_3. \tag{7}$$

This result gives the conclusion of the theorem, i.e., $\Delta \not\vdash \Omega = \Omega_3$, since the axioms defining the lambda theory Δ are contained in $\Sigma + \Omega\Omega_3\Omega = \Omega_3$. In other words, Δ is included into the lambda theory generated by $\Sigma + \Omega\Omega_3\Omega = \Omega_3$. The proof of (7) is obtained as follows. Assume, by the way of contradiction, that $\Sigma + \Omega\Omega_3\Omega = \Omega_3 \vdash \Omega = \Omega_3$. We apply the following version of Jacopini Lemma (see Jacopini [26] and Kuper [31]). There exist closed λ-terms $P_1, \ldots, P_n$, $e_1, \ldots, e_n$ $(n \geq 0)$ such that the following conditions are satisfied (recall the definition of operator * from (1) above):

(i) $e_i \in \{\Omega_3, \Omega\Omega_3\Omega\}$ for every $i = 1, \ldots, n$;

(ii) $\Sigma \vdash \Omega = P_1 e_1$;

(iii) $\Sigma \vdash P_r e_r^* = P_{r+1} e_{r+1}$ for $r = 1, \ldots, n-1$;

(iv) $\Sigma \vdash P_n e_n^* = \Omega_3$.

From (4), (5) and (i)-(iv) above it follows that

$$\Sigma \vdash P_r e_r^* = \Omega, \quad \text{for every } r = 1, \ldots, n,$$

so that from (iv) it follows that

$$\Sigma \vdash \Omega = \Omega_3$$

that contradicts (6). □

The following theorem by Visser as formulated in [3, Thm. 17.1.10] will be used in Thm. 4.3 below.

Theorem 4.2 (Visser [55]) *Let $\mathcal{T} \subsetneq \mathcal{T}'$ be recursively enumerable lambda theories such that $\mathcal{T}' \vdash M = N$ and $\mathcal{T} \not\vdash M = N$. Then there exists a lambda theory $\mathcal{S}$ such that*

$$\mathcal{T} \subsetneq \mathcal{S} \subsetneq \mathcal{T}' \text{ and } \mathcal{S} \not\vdash M = N.$$

Theorem 4.3 *Let $\mathbb{P}$ be the set of real numbers. There exists a family $\mathcal{S} = (\mathcal{S}_r : r \in \mathbb{R})$ of semisensible distinct lambda theories such that $\Delta \subseteq \mathcal{S}_r$ and $\mathcal{S}_r \not\vdash \Omega = \Omega_3$ for all $r \in \mathbb{R}$.*

Proof: Let Π be the consistent lambda theory axiomatized by $\Omega xx = \Omega$ and $\Omega = \Omega_3$. Then $\Delta \subsetneq \Pi$ because $\Pi \vdash \Omega\Omega_3\Omega = \Omega\Omega\Omega = \Omega = \Omega_3$ and $\Delta \not\vdash \Omega = \Omega_3$. By Thm. 4.2 there exists a third lambda theory $\mathcal{S}$ such that $\Delta \subsetneq \mathcal{S} \subsetneq \Pi$ and $\mathcal{S} \not\vdash \Omega = \Omega_3$. Using Thm. 4.2 one can embed the rationals into the recursively enumerable lambda theories included between Δ and Π (see [3, Corollary 17.1.11]), i.e., construct a family $\{\mathcal{S}_r\}_{r \in \mathbb{Q}}$ such that

$$r < r' \rightarrow \mathcal{S}_r \subsetneq \mathcal{S}_{r'} \tag{8}$$

holds for $r, r' \in \mathbb{Q}$. Now define for a real number $r \in \mathbb{R}$ $\mathcal{S}_r = \cup\{\mathcal{S}_q : q < r \text{ and } q \in \mathbb{Q}\}$. This clearly satisfies (8) for $r, r' \in \mathbb{R}$. □

Theorem 4.4 *Let $\mathcal{T}$ be any lambda theory such that $\Delta \subseteq \mathcal{T}$ and $\mathcal{T} \not\vdash \Omega = \Omega_3$. Then every model of $\mathcal{T}$ is a weakly subtractive combinatory algebra.*

Proof: Let $\mathbf{C}$ be a model of $\mathcal{T}$. The interpretation of a closed λ-term M is the element $|M|_{\mathbf{C}}$ of $\mathbf{C}$ (see Section 2.2). For the sake of simplicity, we write directly M for $|M|_{\mathbf{C}}$ when there is no danger of confusion. We have to define a binary term $s(x, y)$ and two constants $0, 1$ satisfying the conditions of Def. 3.1. Define $0 \equiv \Omega$, $1 \equiv \Omega_3$ and $s(x, y) \equiv \Omega xy$. Since $\mathcal{T} \vdash \Omega xx = \Omega$ and $\mathcal{T} \vdash \Omega\Omega_3\Omega = \Omega_3$, then we have that $\mathbf{C} \models \Omega\Omega_3\Omega = \Omega_3$ and $\mathbf{C} \models \lambda x.\Omega xx = \lambda x.\Omega$. This last identity implies $\Omega cc = (\lambda x.\Omega xx)c = (\lambda x.\Omega)c = \Omega$ for all $c \in C$. So, $\mathbf{C}$ is weakly subtractive if Ω and Ω_3 denote different elements of $\mathbf{C}$. This is true because $\mathcal{T} \not\vdash \Omega = \Omega_3$ and then every model of $\mathcal{T}$ distinguishes Ω and Ω_3. □

A *topological model* of lambda calculus is any topological algebra $(\mathbf{C}, \tau)$ such that $\mathbf{C}$ is a λ-model.

Corollary 4.1 *Let $\mathcal{T}$ be any lambda theory such that $\Delta \subseteq \mathcal{T}$ and $\mathcal{T} \not\vdash \Omega = \Omega_3$. If $(\mathbf{C}, \tau)$ is a T_0-topological model of $\mathcal{T}$, then both τ and the specialization order of τ are not co-connected.*

Proof: By Thm. 4.4, Cor. 3.1 and Thm. 3.2. □

A T_0-topological model $(\mathbf{C}, \tau)$ is called a partially ordered λ-model, a *po-model* for short, if τ is the Alexandroff topology defined in Section 2.3. In such a case, the application operator is monotone w.r.t. the specialization order of τ.

Theorem 4.5 *Let $\mathcal{T}$ be any lambda theory such that $\Delta \subseteq \mathcal{T}$ and $\mathcal{T} \not\vdash \Omega = \Omega_3$. Then $\mathcal{T}$ cannot be the theory of a po-model whose specialization order is co-connected.*

Proof: A partial order is co-connected if, and only if, the corresponding Alexandroff topology is co-connected. Then the conclusion follows from the definition of po-model and from Cor. 4.1. □

The models of lambda calculus are classified into *semantics* according to the nature of their representable functions. A semantics is usually constituted by a class of suitable po-models. This last condition is justified by Scott's view of models as sets of sets of observations (or informations) and of computable functions as monotone functions over such sets (see [47]).

Scott's continuous semantics [45] is the class of po-models whose specialization order is a complete partial ordering and the representable functions are all the continuous ones w.r.t. the Scott topology. The graph model semantics (see [46], [19], [37], [38], [10, Section 5.5]) is a subclass of the K-semantics isolated by Krivine (see [30], [10, Section 5.6.2]) within the continuous semantics. The filter model semantics was defined by Coppo, Dezani, Honsell and Longo in [16] (see also [4]) within the continuous semantics.

The stable semantics introduced by Berry [11] is the class of po-models whose specialization order is a DI-domain and the representable functions are all the stable ones.

The strongly stable semantics introduced by Bucciarelly and Ehrhard in [12] is the class of po-models whose specialization order is a DI-domain with coherence and the representable functions are all the strongly stable ones. The hypercoherence semantics introduced by Ehrhard [18] is a subclass of the strongly stable semantics.

Stability and strong stability constitute restrictions of continuity to capture the notion of sequentiality.

The first incompleteness result was given by Honsell and Ronchi della Rocca [25] for the continuous semantics. They proved that the contextual lambda theory induced by the set of essentially closed terms does not admit a continuous model. Following a similar method, Gouy [21] proved the incompleteness of the stable semantics. Other more semantic proofs of incompleteness for the continuous and stable semantics can be found in [7]. Bastonero [6] provides an incompleteness result for the hypercoherence semantics.

Bastonero [6, Section 6], Bastonero-Gouy [7, Section 7] and Berline [10, Section 6.1] conjecture that the strongly stable semantics is also incomplete. We give a positive answer to this open question in the following theorem. We essentially prove that any semantics of lambda calculus based on the concept of approximation of the information is incomplete because of Thm. 4.6(xii) below.

Theorem 4.6 (The Incompleteness Theorem) *The following semantics of the lambda calculus are incomplete. More precisely, there exist $2^{\aleph_0}$ semisensible lambda theories which cannot have a model in the following semantics.*

(i) The graph model semantics.

(ii) The K-semantics.

(iii) The filter model semantics.

(iv) The continuous semantics.

(v) The stable semantics.

(vi) The hypercoherence semantics.

(vii) The strongly stable semantics.

(viii) The po-models with a structure of complete partial ordering.

(ix) The po-models with a structure of meet semilattice.

(x) The po-models with a structure of join semilattice.

(xi) The po-models with a structure of lattice.

(xii) The po-models with a bottom element.

(xiii) The po-models with a top element.

Proof: All the above semantics are given in terms of po-models whose specialization order is co-connected. The conclusion follows from Thm. 4.5 and from Thm. 4.3. □

Recently we have found a simpler proof of the incompleteness theorem based on a more general topological theorem and on the lambda theory axiomatized by the unique identity $\Omega xx = \Omega$. This new proof can be found in the Appendix.

5. Conclusions

We have introduced a new technique to prove the incompleteness of a wide range of lambda calculus semantics (including the strongly stable one, whose incompleteness had been conjectured). Roughly, the technique used for proving that a class $\mathcal{C}$ of models is incomplete is the following:

1. Find a (topological) property P verified by all models in $\mathcal{C}$.
2. Find a lambda theory whose models do not verify P.

To begin with, we remark that the models of lambda calculus based on domains (continuous, stable, strongly stable models in particular) are topological combinatory algebras w.r.t. the Alexandroff topology (the strongest topology whose specialization order is the order of the considered domain), and that they are co-connected (i.e. that the closures of two open sets cannot be disjoint).

Then we define a class of topological algebras which are not co-connected, the weakly subtractive topological algebras.

What has to be shown next is that there exist lambda theories which admit only weakly subtractive combinatory algebras as models. We define a theory Δ and prove that all its models are weakly subtractive, then by standard techniques we get, starting from Δ, a continuum of lambda theories with this same property.

We are working to get a generalization of our incompleteness theorem. The open sets of the Alexandroff topology are closed under arbitrary intersection. This implies that, for every subset V of a poset $(A, \leq)$, there exist a least open set $V\uparrow$, a least closed set $V\downarrow$ and a least clopen (open and closed) set, all of them including V. The minimal clopen sets constitute the partition of the space in connected components. It is possible to prove that every weakly subtractive T_0-topological algebra with the Alexandroff topology admits a clopen set U such that $0 \in U$ and $1 \notin U$. This result implies the incompleteness of every semantics of lambda calculus given in terms of po-models whose Alexandroff topology is connected (recall that a space is connected if there exists no proper clopen set). We conjecture that the semantics of lambda calculus given in terms of po-models whose Alexandroff topology has a finite number of connected components is also incomplete.

Another interesting problem is related to the consistency of the lambda theory $\mathcal{S}$ axiomatized by

$$\Omega xx = \Omega; \quad \Omega x\Omega = x.$$

We conjecture that $\mathcal{S}$ is consistent. A po-model for $\mathcal{S}$ is a subtractive combinatory algebra (see [54]), where Ω is not comparable with any other element in the model (i.e. $a \leq \Omega$ or $\Omega \leq a$ imply $a = \Omega$).

A partial order is trivial if it satisfies $a \leq b$ iff $a = b$. The problem of the incompleteness of the semantics of lambda calculus is also related to the open problem of the order-incompleteness of the lambda theories: does it exist a lambda theory which cannot arise as the theory of any non-trivially partially ordered model? Selinger [50] gave a syntactical characterization, in terms of so-called generalized Mal'cev operators, of the order-incomplete lambda theories. The problem of the order-incompleteness can be stated as follows: does it exist a sequence $M_1, \ldots, M_n$ of closed λ-terms such that the lambda theory $\mathcal{T}_n$, axiomatized by

$$x = M_1xyy;\ M_ixxy = M_{i+1}xyy;\ M_nxxy = y\ (i < n),$$

is consistent? Plotkin and Simpson (see [49]) have shown that $\mathcal{T}_1$ is inconsistent, while Plotkin and Selinger (see [49]) obtained the same result for $\mathcal{T}_2$. It is an open problem whether $\mathcal{T}_n$ $(n \geq 3)$ can be consistent. Order-incompleteness is also related to Plotkin's conjecture (see [39, 49, 50]) about the existence of absolutely unorderable combinatory algebras, where a combinatory algebra is absolutely unorderable if it cannot be embedded in any orderable combinatory algebra.

Acknowledgments

Many thanks to Chantal Berline, Benedetto Intrigila, Simonetta Ronchi della Rocca, Marta Simeoni and the referees for helpful comments and suggestions.

Appendix

We generalize the topological theorem of Section 3 and provide a simpler proof of the incompleteness theorem based on the lambda theory H axiomatized by the unique identity $\Omega xx = \Omega$.

Definition 5.1 *An algebra* **A** *is 3-weakly subtractive if there exist a term* $s(x, y)$ *and two constants* $0, 1$ *in the similarity type of* **A** *such that*

$$s(x, x) = 0; \quad 1 \neq 0; \quad s(1, 0) \neq 0; \quad s(s(1, 0), 0) \neq 0.$$

Definition 5.2 *A 3-weakly subtractive algebra* **A** *is 4-weakly subtractive if*

$$s(s(s(1, 0), 0), 0) \neq 0.$$

Every weakly subtractive algebra is both a 3-weakly and a 4-weakly subtractive algebra.

Theorem 5.1 *Let* $(\mathbf{A}, \tau)$ *be a 3-weakly subtractive* T_0*-topological algebra. Then there exist an open neighbourhood* V *of* 1 *and an open neighbourhood* W *of* 0 *such that* $V \cap W = \emptyset$.

Proof: The proof is divided into claims. Let

$$c \equiv s(1,0); \quad d \equiv s(c,0).$$

Claim 5.1 *There exists an open neighbourhood R of c such that $0 \notin R$.*

By the T_0 hypothesis on τ the elements 0 and d are T_0-separated. We analyse two cases.

(1) There exists a neighbourhood Z of 0 with $d \notin Z$. Then

$$0 = s(c,c) \in Z.$$

By continuity in the second coordinate, there exists an open neighbourhood R of c such that $s(c,R) \subseteq Z$. If $0 \in R$ then $d = s(c,0) \in Z$ that contradicts our hypothesis on Z. Then we have an open neighbourhood R of c such that $0 \notin R$.

(2) There exists a neighbourhood Z of d with $0 \notin Z$. Then

$$d = s(c,0) \in Z.$$

By continuity in the first coordinate, there exists an open neighbourhood R of c such that $s(R,0) \subseteq Z$. If $0 \in R$ then $0 = s(0,0) \in Z$ that contradicts our hypothesis on Z. Then we have an open neighbourhood R of c such that $0 \notin R$.

Claim 5.2 *There exist an open neighbourhood V of 1 and an open neighbourhood W of 0 such that $V \cap W = \emptyset$.*

By Claim 5.1 there exists an open neighbourhood R of c such that $0 \notin R$. From $s(1,0) = c \in R$ and from the continuity of s it follows that there exist two open neighbourhoods V, W of 1 and 0 respectively such that $s(V,W) \subseteq R$. If there is an element $b \in V \cap W$ then $0 = s(b,b) \in R$ that contradicts the hypothesis on R. Then we have $V \cap W = \emptyset$. □

Theorem 5.2 *Let $(\mathbf{A}, \tau)$ be a 4-weakly subtractive T_0-topological algebra. Then there exist an open neighbourhood V of 1 and an open neighbourhood W of 0 such that $\overline{V} \cap \overline{W} = \emptyset$.*

Proof: Let

$$c \equiv s(1,0); \quad d \equiv s(c,0); \quad \epsilon \equiv s(d,0).$$

$\mathbf{A}$ is 3-weakly subtractive in two different ways. It is obvious that the constant 1 satisfies the conditions of Def. 5.1. But the constant c also satisfies the conditions of Def. 5.1:

$$c \neq 0; \quad s(c,0) \neq 0; \quad s(s(c,0),0) \neq 0.$$

Then we can apply Thm. 5.1 to c to get an open neighbourhood V' of c and an open neighbourhood W' of 0 such that $V' \cap W' = \emptyset$.

Since s is continuous and $s(1,0) = c \in V'$, there exist two other open sets V and W containing 1 and 0, respectively, such that $s(V,W) \subseteq V'$. The sets V and W will be the right sets for the conclusion of the theorem. Since s is continuous the pre-image of $\overline{V'}$ under the map s is closed. From $s(V,W) \subseteq V' \subseteq \overline{V'}$ the pre-image of $\overline{V'}$, that is closed, contains $V \times W$, so $s(\overline{V},\overline{W}) \subseteq \overline{V'}$.

We now prove that $\overline{V} \cap \overline{W} = \emptyset$. Assume, by the way of contradiction, that there is $f \in \overline{V} \cap \overline{W}$. Since $s(\overline{V},\overline{W}) \subseteq \overline{V'}$ it follows that $0 = s(f,f) \in \overline{V'}$. But by definition of closure of a set this is possible only if for every open neighbourhood Z of 0, we have that $Z \cap V' \neq \emptyset$. But this contradicts our initial choice of V' and W' as two open neighbourhoods of c and 0 respectively with empty intersection. □

Consider the semisensible lambda theory Π axiomatized by

$$\Omega x x = \Omega.$$

Define

$$t_0 \equiv \Omega_3; \quad t_{n+1} \equiv \Omega(t_n)\Omega.$$

Theorem 5.3 *We have:*

$$\Pi \not\vdash t_n = \Omega \text{ for all } n.$$

Proof: Let $\to_\Pi$ be the following reduction rule:

$$\Omega M N \to_\Pi \Omega \tag{9}$$

for every M and N such that $\Pi \vdash M = N$. The reflexive closure of $\to_\Pi$ satisfies the diamond property, and the relations $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_\Pi$ commute. Then the reduction rule $\to_{\beta\Pi} = \to_\beta \cup \to_\Pi$ is Church-Rosser by the Hindley-Rosen Lemma (see Berarducci-Intrigila [9, Thm. 3.4] and Barendregt [3, Prop. 3.3.5]).

Then we prove that Π is the lambda theory generated by conversion $\cong_{\beta\Pi}$ from $\to_{\beta\Pi}$, i.e.,

$$\Pi \vdash M = N \text{ iff } M \cong_{\beta\Pi} N. \tag{10}$$

Since $\Omega M N \to_\Pi \Omega$ iff $\Pi \vdash M = N$, then it is obvious that $M \cong_{\beta\Pi} N$ implies $\Pi \vdash M = N$. For the opposite direction, it is sufficient to consider that $\Omega x x \to_\Pi \Omega$ for the unique axiom $\Omega x x = \Omega$ of Π.

We now prove by induction that $\Pi \not\vdash t_n = \Omega$. First we have that $\Pi \not\vdash t_0 = \Omega$ because $t_0 \equiv \Omega_3$ and Ω do not have a common $\beta\Pi$-reduct. By the way of contradiction, assume $\Pi \vdash t_{n+1} = \Omega$, so that $t_{n+1} \equiv \Omega(t_n)\Omega \cong_{\beta\Pi} \Omega$. Then there exists a reduction $\Omega(t_n)\Omega \twoheadrightarrow_{\beta\Pi} \Omega$. This is possible only if $\Omega(t_n)\Omega$ is a Π-redex i.e. if $\Pi \vdash t_n = \Omega$. But this contradicts the induction hypothesis. □

Theorem 5.4 *Every model of the lambda theory Π is a 4-weakly subtractive combinatory algebra.*

Proof: Let $\mathbf{C}$ be a model of Π. We have to define a binary term $s(x,y)$ and two constants $0,1$ satisfying the conditions of Def. 5.2. Define $0 \equiv \Omega$, $1 \equiv \Omega_3$ and $s(x,y) \equiv \Omega xy$. The proof of the theorem is now similar to that of Thm. 4.4 and it is omitted. □

Theorem 5.5 *The lambda theory* Π *cannot be the theory of a po-model whose Alexandroff topology is co-connected.*

Proof: It follows from Thm. 5.4 and from Thm. 5.2. □

Corollary 5.1 *The lambda theory* Π, *axiomatized by* $\Omega xx = \Omega$, *cannot have a model in the semantics specified in Thm. 4.6.*

References

[1] S. Abramsky, "Domain theory in logical form", *Annals of Pure and Applied Logic*, **51** (1991), pp. 1-77

[2] S. Abramsky and C.H.L. Ong, "Full abstraction in the lazy λ-calculus", *Information and Computation*, **105** (1993), pp. 159-267

[3] H.P. Barendregt, *The lambda calculus: Its syntax and semantics*, Revised edition, Studies in Logic and the Foundations of Mathematics **103**, North-Holland Publishing Co., Amsterdam (1984)

[4] H.P. Barendregt, M. Coppo and M. Dezani-Ciancaglini, "A filter model and the completeness of type assignment", *Journal of Symbolic Logic*, **48** (1983), pp. 931-940

[5] O. Bastonero, *Modéles fortement stables du λ-calcul et résultats d'incomplétude*, Thése, Université de Paris 7 (1996)

[6] O. Bastonero, *Equational incompleteness and incomparability results for λ-calculus semantics*, manuscript

[7] O. Bastonero and X. Gouy, "Strong stability and the incompleteness of stable models of λ-calculus", *Annals of Pure and Applied Logic*, **100** (1999), pp. 247-277

[8] W. Bentz, "Topological implications in varieties", *Algebra Universalis*, **42** (1999), pp. 9-16

[9] A. Berarducci and B. Intrigila, "Some new results on easy lambda-terms", *Theoretical Computer Science* **121** (1993), pp. 71-88

[10] C. Berline, "From computation to foundations via functions and application: The λ-calculus and its webbed models", *Theoretical Computer Science* **249** (2000), pp. 81-161

[11] G. Berry, "Stable models of typed lambda-calculi", Proc. 5th Int. Coll. on Automata, Languages and Programming, LNCS vol.62, Springer-Verlag (1978)

[12] A. Bucciarelli and T. Ehrhard, "Sequentiality and strong stability", Sixth Annual IEEE Symposium on Logic in Computer Science (1991), pp. 138-145

[13] S. Burris and H.P. Sankappanavar, *A course in universal algebra*, Springer-Verlag, Berlin (1981)

[14] J.P. Coleman, "Separation in topological algebras", *Algebra Universalis*, **35** (1996), pp. 72-84

[15] J.P. Coleman, "Topological equivalents to n-permutability", *Algebra Universalis*, **38** (1997), pp. 200-209

[16] M. Coppo, M. Dezani-Ciancaglini, F. Honsell and G. Longo, "Extended type structures and filter λ-models", Logic Colloquium'82, Elsevier Science Publishers (1984), pp. 241-262

[17] H.B. Curry and R. Feys, *Combinatory Logic*, Vol. I, North-Holland Publishing Co., Amsterdam (1958)

[18] T. Ehrhard, "Hypercoherences: a strongly stable model of linear logic", *Mathematical Structures in Computer Science*, **2** (1993), pp. 365-385

[19] E. Engeler, "Algebras and combinators", *Algebra Universalis*, **13** (1981), pp. 389-392

[20] J.Y. Girard, "The system F of variable types, fifteen years later", *Theoretical Computer Science*, **45** (1986), pp. 159-192

[21] X. Gouy, *Etude des théories équationnelles et des propriétés algébriques des modéles stables du λ-calcul*, Thèse, Université de Paris 7 (1995)

[22] G. Grätzer, *Universal Algebra*, Second edition, Springer-Verlag, New York (1979)

[23] H.P. Gumm, "Topological implications in n-permutable varieties", *Algebra Universalis*, **19** (1984), pp. 319-321

[24] F. Honsell and M. Lenisa, "Final semantics for untyped λ-calculus", LNCS 902, Springer-Verlag, Berlin (1995), pp. 249-265

[25] F. Honsell and S. Ronchi della Rocca, "An approximation theorem for topological λ-models and the topological incompleteness of λ-calculus", *Journal Computer and System Science*, **45** (1992), pp. 49-75

[26] C. Jacopini, "A condition for identifying two elements in whatever model of combinatory logic", LNCS 37 (C. Bohm ed.), Springer-Verlag, Berlin (1975)

[27] P.T. Johnstone, *Stone Spaces*, Cambridge University Press (1982)

[28] R. Kerth, "Isomorphism and equational equivalence of continuous lambda models", *Studia Logica*, **61** (1998), pp. 403-415

[29] R. Kerth, "On the construction of stable models of λ-calculus", *Theoretical Computer Science* (to appear)

[30] J.L. Krivine, *Lambda-Calcul, types et modèles*, Masson, Paris (1990)

[31] J. Kuper, "On the Jacopini technique", *Information and Computation*, **138** (1997), pp. 101-123

[32] R.N. McKenzie, G.F. McNulty and W.F. Taylor, *Algebras, Lattices, Varieties, Volume I*, Wadsworth Brooks, Monterey, California, (1987)

[33] A.R. Meyer, "What is a model of the lambda calculus?", *Information and Control*, **52**, (1982), pp. 87-122

[34] D. Pigozzi and A. Salibra, "Lambda abstraction algebras: representation theorems", *Theoretical Computer Science*, **140** (1995), pp. 5-52

[35] D. Pigozzi and A. Salibra, "The abstract variable-binding calculus", *Studia Logica*, **55** (1995), pp. 129-179

[36] D. Pigozzi and A. Salibra, "Lambda abstraction algebras: coordinatizing models of lambda calculus", *Fundamenta Informaticae* **33** (1998), pp. 149-200

[37] G.D. Plotkin, "A set theoretic definition of application", Memorandum MIP-R-95, University of Edinburgh (1972)

[38] G.D. Plotkin, "Set-theoretical and other elementary models of the λ-calculus", *Theoretical Computer Science*, **121** (1993), pp. 351-409

[39] G.D. Plotkin, "On a question of H. Friedman", *Information and Computation*, **126** (1996), pp. 74-77

[40] A. Salibra, "Nonmodularity results for lambda calculus", *Fundamenta Informaticae* (to appear)

[41] A. Salibra, "On the algebraic models of lambda calculus", *Theoretical Computer Science*, **249** (2000), pp. 197-240

[42] A. Salibra and R. Goldblatt, "A finite equational axiomatization of the functional algebras for the lambda calculus", *Information and Computation*, **148** (1999), pp. 71-130

[43] M. Schönfinkel,"Uber die bausteine der Mathematischen Logik", *Mathematischen Annalen* (english translation in J. van Heijenoort ed.'s book"From Frege to Gödel, a source book in Mathematical Logic, 1879-1931", Harvard University Press, 1967), **92** (1924), pp. 305-316

[44] D.S. Scott, "Some ordered sets in computer science", In: Ordered sets (I. Rival ed.), Proc. of the NATO Advanced Study Institute (Banff, Canada), Reidel (1981), pp. 677-718

[45] D.S. Scott, "Continuous lattices", In: Toposes, Algebraic geometry and Logic (F.W. Lawvere ed.), LNM 274, Springer-Verlag (1972), pp. 97-136

[46] D.S. Scott, "Data types as lattices", *SIAM J. Computing*, **5** (1976), pp. 522-587

[47] D.S. Scott, "Lambda calculus: some models, some philosophy", The Kleene Symposium (J. Barwise, H.J. Keisler, and K. Kunen eds.), Studies in Logic 101, North-Holland (1980)

[48] D.S. Scott and C. Gunter, "Semantic domains", Handbook of Theoretical Computer Science, North-Holland, Amsterdam (1990)

[49] P. Selinger, *Functionality, polymorphism, and concurrency: a mathematical investigation of programming paradigms*, PhD thesis, University of Pennsylvania (1997)

[50] P. Selinger, "Order-incompleteness and finite lambda models", Eleventh Annual IEEE Symposium on Logic in Computer Science (1996)

[51] L.A. Steen and J.A. Seebach, Jr., *Counterexamples in topology*, Springer-Verlag (1978)

[52] W. Taylor "Varieties of topological algebras", *Austral. Math. Soc.*, **23** (1977) pp. 207-241

[53] W. Taylor, "Varieties obeying homotopy laws", *Canad. Journal Math.*, **29** (1977), pp. 498-527

[54] A. Ursini, "On subtractive varieties, I", *Algebra Universalis*, **31** (1994), pp. 204-222

[55] A. Visser, "Numerations, λ-calculus and arithmetic", To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism (J.R. Hindley and J.P. Seldin eds.), Academic Press, New York (1980), pp. 259-284

Session 9

Relating levels of the mu-calculus hierarchy and levels of the monadic hierachy

David Janin Giacomo Lenzi
LaBRI
Université de Bordeaux I - ENSERB
351 cours de la Libération,
F-33 405 Talence cedex
{janin|lenzi}@labri.u-bordeaux.fr

Abstract

As already known [14], the mu-calculus [17] is as expressive as the bisimulation invariant fragment of monadic second order Logic (MSO). In this paper, we relate the expressiveness of levels of the fixpoint alternation depth hierarchy of the mu-calculus (the mu-calculus hierarchy) with the expressiveness of the bisimulation invariant fragment of levels of the monadic quantifiers alternation-depth hierarchy (the monadic hierarchy).

From van Benthem's result [3], we know already that the fixpoint free fragment of the mu-calculus (i.e. polymodal Logic) is as expressive as the bisimulation invariant fragment of monadic Σ_0 (i.e. first order logic). We show here that the ν-level (resp. the $\nu\mu$-level) of the mu-calculus hierarchy is as expressive as the bisimulation invariant fragment of monadic Σ_1 (resp. monadic Σ_2) and we show that no other level Σ_k for $k > 2$ of the monadic hierarchy can be related similarly with any other level of the mu-calculus hierarchy.

The possible inclusion of all the mu-calculus in some level Σ_k of the monadic hierarchy, for some $k > 2$, is also discussed.

1 Introduction

The propositional modal fixpoint calculus (or mu-calculus for short) introduced by Kozen [17] is considered in this paper. The mu-calculus was initially introduced as a specification formalism for processes modeled as states in transition systems.

However, using the mu-calculus as a logic of processes has a major drawback : the model-checking problem, which is to decide if a (finite) model (given as input) satisfies a formula (also given as input), remains somehow difficult. More precisely, the best model checking algorithms known so far - see [16] for the lastest development - have (time) complexity $O((mn)^{\lceil d/2 \rceil + 1})$ where m is the size of the input graph, n is the size of the formula and d is the fixpoint alternation-depth of the formula which depends on the input formula. Moreover the restriction to mu-calculus formulas with a bounded fixpoint alternation-depth is (theoretically) not an issue because it also strictly reduces the expressive power of the logic. Indeed, Bradfield [4] and, in some weaker sense, Lenzi [18], prove that the hierarchy induced by the fixpoint alternation-depth (the mu-calculus hierarchy) is strict.

In practice, temporal logics [6], which all belong to low levels of the alternation depth hierarchy, are often preferred to the full mu-calculus since in that case the model checking problem has a low degree polynomial (even linear) time complexity.

It is also known that the model-checking problem belongs to $NP \cap$co-NP [15]. From Fagin's famous correspondence between the class NP and the existential fragment of second order logic [7], this upper bound tells us that all mu-calculus formulas belongs to the level $\Sigma_1 \cap \Pi_1$ of the second order quantifier alternation hierarchy.

Since all mu-calculus formulas can be translated into monadic second order logic (MSO) one may ask whether similar *descriptive complexity results* are available for the monadic quantifier alternation hierarchy (the monadic hierarchy) which is known to be strict (even over finite models as shown by Matz and Thomas [20]). More precisely, since the mu-calculus is as expressive as (or equivalent to) the bisimulation invariant fragment of MSO [14], one may ask whether the full mu-calculus or any level of the mu-calculus hierarchy is equivalent to the bisimulation invariant fragment of some level of the monadic hierarchy.

Van Benthem [3] already shows that the fixpoint free fragment of the mu-calculus (i.e. Polymodal Logic also called Hennessy-Milner logic among computer scientists) is equivalent to the bisimulation invariant fragment of

0-7695-1281-X/01 $10.00 © 2001 IEEE

Levels of the mu-calculus	*Levels of the monadic hierarchy*	*Reference*
Mu-calculus	Monadic Second Order Logic	Janin-Walukiewicz 1996
Polymodal Logic	FOL	Van Benthem 1976
ν-level of the mu-calculus	monadic Σ_1	shown here
$\nu\mu$-level of the mu-calculus	monadic Σ_2	shown here
Properties (all[1]) of arbitrary levels	monadic Σ_3	shown here

Figure 1. Correspondance between levels of the mu-calculus hierarchy and levels of the bisimulation invariant fragment of the monadic hierarchy

monadic Σ_0 (i.e. FOL).

Here, we complete the picture showing that :

Theorem 1.1 *The ν-level (resp. the μ-level) of the mu-calculus hierarchy is equivalent to the bisimulation invariant fragment of the level Σ_1 (resp. Π_1) of the monadic hierarchy.*

and

Theorem 1.2 *The $\nu\mu$-level (resp. the $\mu\nu$-level) of the mu-calculus hierarchy is equivalent to the bisimulation invariant fragment of the level Σ_2 (resp. Π_2) of the monadic hierarchy.*

From Arnold's proof of the strictness of the mu-calculus hierarchy [2], we also show that :

Theorem 1.3 *For each integer $k > 2$ there exists a bisimulation invariant formula of monadic Σ_3 that does not belong to the kth level of the mu-calculus hierarchy.*

In other words, no other equivalence similarly relates levels of the mu-calculus hierarchy with levels of the monadic hierarchy.

The question whether the mu-calculus is equivalent to the bisimulation invariant fragment of monadic Σ_k, for some integer $k > 2$, remains, strictly speaking, open. However, the following theorem, which is a consequence of the work of Courcelle [5], shows that, on a quite general class of graphs (or the class of all graphs[1]), this is already true with monadic Σ_3.

Theorem 1.4 *Over the class of graphs of bounded degree (or bounded tree-width) all mu-calculus formulas can be translated into monadic Σ_3 formulas.*

Figure 1 above summarizes all these results. One must be aware that, for these results, we are considering arbitrary finite *and* infinite models. Rosen [28] shows that van Benthem's result still holds over finite models only. All other statements mentioned in Figure 1 are open problems over finite models.

[1]provided, as in MS_2 in [5], quantification over edges is available !

Allthough these new results essentially have a theoretical flavor they can also be seen as a general toolkit to analyse, from syntax, the model-checking complexity of logics of programs. Indeed, most logics of programs are (implicitly defined as) particular fragments of the bisimulation invariant fragment of MSO. The result above says that, as soon as these logics can be translated into monadic Δ_1 (resp. monadic Δ_2) then the model checking complexity is linear (resp. quadratic) in the size of the input program.

Related works

The study of various bisimulation invariant fragments of logical formalisms leads to some other results.

Following Hafer and Thomas [10] logical characterization of CTL* over the binary tree, Moller and Rabinovich [21] obtain a similar characterization of CTL* over arbitrary trees : CTL* is as expressive as the bisimulation invariant fragment of MSO over trees with path quantifiers instead of general set quantifiers.

With a more expressive language than the mu-calculus, Grädel, Hirsch and Otto show the expressive completeness of the guarded fixpoint calculus w.r.t. the bisimulation invariant fragment of guarded second order logic [9].

Over finite models, Otto gives a fixpoint characterization of bisimulation invariant PTIME [25].

In his PhD thesis [11], Hollenberg also characterizes the bisimulation invariant fragment of MSO via *bisimulation-quantifiers* [8]. It is an open question whether his approach extends to the bisimulation invariant fragment of monadic Σ_1 or monadic Σ_2.

Investigating bisimulation invariance inside MSO also leads to apply works on MSO over trees. The pioneering works of Rabin [26][27] on the monadic second order theory of the binary tree (S2S) are obviously relevant here. Also the many automata characterization of various mu-calculi over trees which starts in the early 80's with the results of Niwinski [24] or Street and Emerson [32] among others are fundamental. In this paper, we use one of the last and most achieved extension of these techniques and results obtained by Walukiewicz [33].

Note however, Theorems 1.1 and 1.2 are not immediate consequences of these results.

For the analysis of bisimulation invariance inside monadic Σ_1, the restriction to trees is even misleading since, with properties definable in monadic Σ_1, bisimulation invariance over trees is less restrictive than bisimulation invariance over arbitrary graphs. Indeed, the monadic Σ_1 formula $\exists x p(x)$, although bisimulation invariant over trees, would mean, as a bisimulation invariant property over graphs, that there is a directed path from a distinguished vertex (the root of the graph) to some vertex x where p holds. This property is at least as difficult to express as directed reachability which, as shown by Ajtai and Fagin [1], is not expressible in monadic Σ_1.

For the analysis of bisimulation invariance inside monadic Σ_2, it is true that bisimulation invariance over trees or graphs coincides. But then, there is no real characterizations of FOL or monadic Σ_1 logic of trees so no simple inductive proof is available. To prove Theorem 1.2, we shall extend to all trees a new similar result of Lenzi [19], reproved by Skurczyński [31] in a more automata theoretical way, which says that, on the binary tree, languages definable in monadic Σ_2 are exactly the languages recognizable by tree automata with Büchi conditions.

Overview

The paper is organized as follows. First we recall the definition of bisimulation equivalence. Then, in relation with it, we present the notions of κ-expansions which provide, in some sense, canonical representatives of bisimulation equivalences classes of graphs.

In the third part, we recall the definitions of Monadic Second Order Logic and the modal and counting mu-calculus. We also recall most of the known results relating these languages.

In the fourth part, we give a definition of tree automata which, with various acceptance criteria, will constitute the main technical tools to prove our results.

In the fifth and sixth parts, bisimulation invariance in monadic Σ_1 and in monadic Σ_2 are analyzed. Sketch of proofs for Theorem 1.1 and Theorem 1.2 are given.

In the last part, the case of levels Σ_k for $k > 2$ is considered and Theorem 1.3 and Theorem 1.4 are proved.

Acknowledgement

Thanks to André Arnold and Igor Walukiewicz for many stimulating and helpful discussions on this topic. Thanks to Mike Robson for his help writting this final version.

2 Graphs, Bisimulation and Expansion

We recall here the notions of transition systems, bisimulation equivalence and expansion of transition systems. Since a transition system is simply a directed graph with a distinguished vertex called its source or root, we use in the following the vocabulary of (directed) graphs.

Also, in order to simplify statements and proofs, we only consider here unlabeled directed graphs (built over a single binary relation symbol). One can check that all the results presented here can easily be generalized to (finitely) labeled directed graphs, i.e. graphs built over a finite set of binary relation symbols.

Let *Prop* be a set of unary predicate symbols and let R be a binary relation symbol. A *graph with a root*, simply called *graph* in the sequel, is a tuple:

$$M = \langle S^M, r^M, R^M, \{p^M\}_{p \in Prop} \rangle$$

with a set S^M of *vertices*, a *root* $r^M \in S^M$, a binary *successor relation* $R^M \subseteq S^M \times S^M$ and for each $p \in Prop$, a subset $p^M \subseteq S^M$.

Graphs M and N are called *bisimilar* when there exists a relation $R \subseteq S^M \times S^N$, called a *bisimulation relation*, such that $(r^M, r^N) \in R$ and for every $(s,t) \in R$ and $p \in Prop$, $s \in p^M$ iff $t \in p^N$, and whenever $(s,s') \in R^M$ for some s', then there exists t' such that $(t,t') \in R^N$ and $(s',t') \in R$, and whenever $(t,t') \in R^N$ for some t', then there exists s' such that $(s,s') \in R^M$ and $(s',t') \in R$.

Given any set κ (disjoint from S^M), a *κ-indexed path* in M is a non empty finite or infinite word $w \in S^M.(\kappa.S^M)^\infty$ such that whenever $w = u.s.k.s'.v$ with $u \in (S^M.\kappa)^*$, $s \in S^M$, $k \in \kappa$, $s' \in S^m$ and $v \in (\kappa.S^M)^\infty$ one has $(s,s') \in R^M$. The length $|w|$ of κ-index path w is defined as the number of occurrences of elements of S^M in w, e.g. when $w = s_0.k_1.s_1.\cdots.k_n.s_n$ we put $|w| = n+1$. In this case, we say s_0 is the source of w, s_n is the target of w and w is a (κ-indexed) path from s_0 to s_n.

Remark that (up to isomorphism) the notion of κ-indexed path only depends on the cardinality of κ. In particular, when κ is a singleton, κ-indexed paths are nothing but the usual (directed) paths in a graph.

The *κ-expansion* $T^\kappa(M)$ of system M is defined as follows : set $S^{T^\kappa(M)}$ is the set of all finite κ-indexed paths of M with root r^M, the root $r^{T^\kappa(M)}$ equals r^M, relation $R^{T^\kappa(M)}$ is the set of all pairs of the form $(u.s, u.s.k'.s') \in S^{T^\kappa(M)} \times S^{T^\kappa(M)}$ with $u \in (S^M.\kappa)^*$, s and $s' \in S^M$ and $k' \in \kappa$ such that $(s,s') \in R^M$, and, for any $p \in Prop$, $p^{T^\kappa(M)}$ is the set of all κ-indexed path of the form $u.s \in S^{T^\kappa(M)}$ with $u \in (S^M.\kappa)^*$ and $s \in p^M$.

Any κ-expansion is a tree. Moreover, when κ is a singleton, the κ-expansion of M, from now on denoted by $T(M)$, is nothing but what is usually called the unwinding or un-

raveling of graph M from its root r^M. Vertices of $T(M)$ are all finite paths from the root.

When M is a tree, i.e. when M and $T(M)$ are isomorphic, we shall use the notation $\leq^M$ for the *order relation* induced by the tree-structure of M, i.e. relation $\leq^M$ is the reflexive and transitive closure of relation R^M.

The notion of κ-expansion gives in some sense canonical representatives of equivalence classes under bisimulation as illustrated by the following fact.

Fact 2.1 *For any infinite set κ and for any graphs M and N of cardinality at most $|\kappa|$, M and N are bisimilar iff $T^\kappa(M)$ and $T^\kappa(N)$ are isomorphic.*

3 First order and monadic second order logic and the propositional μ-calculus

In this section we define first order logic (FO) and monadic second order logic (MSO) and two variants of the propositional μ-calculus [17]. All logics are interpreted over transition systems. Note that a transition system M, as defined above, is a FO-structure with domain $dom(M) = S^M$ on the vocabulary $\{r, R\} \cup Prop$ with r a constant symbol standing for the root, R a binary relation symbol and $Prop$ a set of unary relation symbols.

3.1 FO and MSO

Let $var = \{x, y, \cdots\}$ and $Var = \{X, Y, \cdots\}$ be respectively some disjoint sets of first order and monadic second order variable symbols.

First order logic over the vocabulary $\{r, R\} \cup Prop$ can be defined as follows. The set of FO formulas is the smallest set containing formulas $p(t)$, $t = t'$, $R(t, t')$, $X(t)$ for $p \in Prop$, $X \in Var$ and $t \in var \cup \{r\}$ and closed under negation $\neg$, disjunction $\vee$, conjunction $\wedge$ and existential $\exists$ and universal $\forall$ quantifications over FO variables.

Monadic second order logic over the vocabulary $\{r, R\} \cup Prop$ can be defined as follows. The set of MSO formulas is the smallest set containing all FO formulas and closed under negation $\neg$, disjunction $\vee$, conjunction $\wedge$ and existential $\exists$ and universal $\forall$ quantifications over set variables.

For any MSO formula, we use the notation $\varphi(x_1, \cdots, x_m, X_1, \cdots, X_n)$ for the formula φ with free first order variables among $\{x_1, \cdots, x_m\}$ and free set variables among $\{X_1, \cdots, X_n\}$. For any graph M, any elements $s_1, \ldots, s_m \in S^M$, any sets $S_1, \ldots, S_n \subseteq S^M$, we use the notation

$$M \models \varphi(s_1, \cdots, s_m, S_1, \cdots, S_n)$$

to say that formula φ is true in M, or M satisfies φ, under the interpretation of each FO variable x_i by the vertex s_i and each set variable X_j by the set S_j. We do not repeat here the definition of this satisfaction relation.

A class $\mathcal{C}$ of graph is said *MSO definable* when there exists a sentence $\varphi \in MSO$, i.e. a formula with no free variable, such that $M \in \mathcal{C}$ iff $M \models \varphi$. A class $\mathcal{C}$ of transition systems is *bisimulation closed* (resp. *closed under unwinding*) if whenever $M \in \mathcal{C}$ and M' is bisimilar to M then $M' \in \mathcal{C}$ (resp. if for any M, $M \in \mathcal{C}$ iff $T(M) \in \mathcal{C}$). A sentence φ is *bisimulation invariant* (resp. *unwinding invariant*) if the class of transition systems it defines is bisimulation closed (resp. closed under unwinding). Remark that bisimulation invariance implies unwinding invariance since any graph M is bisimilar to its unwinding $T(M)$.

The notion of bisimulation invariance (or unwinding invariance) extend to arbitrary formula $\varphi(X_1, \cdots, X_n)$ with no free FO variable considering graphs over the set of predicate symbols $Prop' = Prop \cup \{X_1, \cdots, X_n\}$. Since fixpoint formulas, which we will consider later, may have free set variables, we shall implicitly consider this extension of graph to $Prop'$ whenever there is no ambiguity.

Finally, the monadic quantifier alternation-depth hierarchy is defined as follows. The first level $\Sigma_0 = \Pi_0$ is defined as the set of all formulas of first order logic. Then , for each integer k, level Σ_{k+1} (resp. level Π_{k+1}) is defined as the set of all formulas of the form $\exists X_1 \cdots \exists X_n \varphi$ with $\varphi \in \Pi_k$ (resp. $\forall X_1 \cdots \forall X_n \varphi$ with $\varphi \in \Sigma_k$). The bisimulation invariant (resp. unwinding invariant) fragment of the level Σ_k of MSO formulas is defined as the set of all bisimulation invariant (resp. unwinding invariant) formulas of Σ_k with no free first order variables.

3.2 Modal and counting μ-calculus

The set of the modal μ-calculus formulas is the smallest set containing $Prop \cup Var$ which is closed under negation, disjunction and the following formation rules:

- if α is a formula then $\Diamond\alpha$ and $\Box\alpha$ are formulas,
- if $\alpha(X)$ is a formula and X occurs only positively (i.e. under even number of negations) in $\alpha(X)$ then $\mu X.\alpha(X)$ and $\nu X.\alpha(X)$ are formulas.

The set of counting μ-calculus formulas is defined as above replacing standard modalities $\Diamond$ and $\Box$ by counting modalities $\Diamond_k$ and $\Box_k$ for any integer k.

We use the same convention as for MSO with free set variables, i.e. we denote by $\alpha(X_1, \cdots, X_n)$ a formula with free variables among $\{X_1, \cdots, X_n\}$. For convenience, we may also omit these free set variables in formula α considering then implicitly that graphs have been built over the set of unary predicate symbols $Prop' = Prop \cup \{X_1, \cdots, X_n\}$. In the sequel, we call *fixpoint formula* any formula of the modal or counting μ-calculus.

Atomic formulas :	$\varphi_p = p(r),\ \varphi_X = X(r),$
Boolean connectives :	$\varphi_{\alpha\wedge\beta} = \varphi_\alpha \wedge \beta,\ \varphi_{\alpha\vee\beta} = \varphi_\alpha \vee \beta$ and $\varphi_{\neg\alpha} = \neg\varphi_\alpha$
Modalities :	$\varphi_{\Diamond\alpha} = \exists z\ R(r,z) \wedge \varphi_\alpha[z/r],\ \varphi_{\Box\alpha} = \forall z\ R(r,z) \Rightarrow \varphi_\alpha[z/r]$
Counting modalities :	$\varphi_{\Diamond_k\alpha} = \exists z_1,\cdots,z_k\ \mathit{diff}(z_1,\cdots,z_k) \wedge \bigwedge_{i\in[1,k]} R(r,z_i) \wedge \varphi_\alpha[z_i/r]$ and $\varphi_{\Box_k\alpha} = \forall z_1,\cdots,z_k\ (\mathit{diff}(z_1,\cdots,z_k) \wedge \bigwedge_{i\in[1,k]} R(r,z_i)) \Rightarrow \bigvee_{i\in[1,k]} \varphi_\alpha[z_i/r]$
Fixpoints :	$\varphi_{\mu X.\alpha(X)} = \forall X(\forall z\varphi_{\alpha(X)}[z/r] \Rightarrow X(z)) \Rightarrow X(r)$ and $\varphi_{\nu X.\alpha(X)} = \exists X(\forall z X(r) \Rightarrow \varphi_{\alpha(X)}[z/r]) \wedge X(r)$

Figure 2. Semantics of fixpoint formulas

The meaning of a fixpoint formula α in a transition system M can be defined as an MSO formula φ_α with no free first order variables and with the same free set variables. The inductive definition of φ_α is described in Figure 2 below. In this figure, $\mathit{diff}(z_1,\cdots,z_k)$ is the quantifier free FO formula stating that $z_i \neq z_j$ for all $i \neq j$, α and β are arbitrary formulas, k is any integer, X any second order variable, and z, z_1, ..., z_k any FO variables. Formula $\varphi_\alpha[z/r]$ is the formula obtained from φ_α by replacing any occurrence of r by z, provided FO variable z has been chosen in such a way it is never captured by a FO quantification during this replacement.

Remark that one can choose FO variables in such a way that, for any modal mu-calculus formulas α, formula φ_α is defined using at most two FO variables and, for any counting mu-calculus formulas α, φ_α is defined using at most $k+1$ variables where k is the greatest integer such that modality $\Diamond_k$ or $\Box_k$ occurs in α.

For any fixpoint formula α, we shall write $M \models \alpha$ when $M \models \varphi_\alpha$. We say that an MSO formula φ is equivalent to a fixpoint formula α when $\models \varphi_\alpha \Leftrightarrow \varphi$.

The following fact follows from the above definitions :

Fact 3.1 *For any fixpoint formula, if α is a modal (resp. counting) mu-calculus formula then φ_α is bisimulation invariant (resp. unwinding invariant).*

The following theorems show that the above invariance properties characterize in some sense the expressive power of these fixpoint calculi.

Theorem 3.2 (from Walukiewicz [33]) *A MSO sentence is invariant under unwinding iff it is equivalent to some counting mu-calculus formula.*

and

Theorem 3.3 (Janin-Walukiewicz [14]) *A MSO sentence is invariant under bisimulation iff it is equivalent to some modal mu-calculus formula.*

Finally, the (modal or counting[2]) fixpoint alternation-depth hierarchy defined as follows. The first level $N_0 = M_0$ is defined as the set of all (modal or counting) fixpoint free formula with negation only applied to propositional constants of *Prop*. Then, for each integer k, level N_{k+1} (resp. level M_{k+1}) is defined as the closure of $N_k \cup M_k$ under disjunction, conjunction, substitution - provided no free variable becomes bounded during the substitution process - and greatest fixpoint construction (resp. least fixpoint construction). In the sequel, we shall also call ν-level (resp. μ-level) or $\nu\mu$-level (resp. $\mu\nu$-level) of the fixpoint hierarchies, the level N_1 (resp. M_1) or N_2 (resp. M_2).

Theorem 3.4 (Bradfield [4]) *For each integer k there is a modal mu-calculus formula $\alpha \in N_k$ which is not equivalent to any modal mu-calculus formula in $N_{k'}$ with $k' < k$.*

Arnold [2] shows that the above result still holds restricted to the binary tree. From this stronger result we also have :

Theorem 3.5 (From Arnold [2]) *For each integer k there is a counting mu-calculus formula $\alpha \in N_k$ which is equivalent to no counting mu-calculus formula in $N_{k'}$ with $k' < k$.*

Proof. Observe first that the binary tree is definable in the counting mu-calculus with a formula of N_1. Moreover, over the binary tree (with distinct left and right successors) the counting and the modal mu-calculus are - level by level - equally expressive. So Arnold's result extends to the counting fixpoint hierarchy. □

4 Infinite tree automata

We define here tree automata that characterize the expressive power of the two mu-calculi defined above. Although the main ideas and proof techniques go back to, at least, the work of Streett and Emerson on the mu-calculus [32], it took some times for these techniques to

[2] depending on the modalities one allows

be really understood and generalized to wider settings than the non emptiness or the model checking problem for the modal mu-calculus alone. In this section, we more or less follow Walukiewicz's general approach [33].

In the sequel, the *alphabet* Σ is defined as the powerset $\mathcal{P}(Prop)$ of *Prop*. The intuition behind this is that a vertex x in a tree M is labeled by the "letter" $\lambda(x) \in \Sigma$ defined as the set $\lambda(x) = \{p \in Prop : x \in p^M\}$.

An *alternating counting tree-automaton* is a tuple

$$\mathcal{A} = \langle Q, \Sigma, q_0, \Omega, \delta \rangle$$

for a finite set of *states* Q, the finite *alphabet* Σ, an *initial state* $q_0 \in Q$, a *parity index function* $\Omega : Q \to I\!N$ and the *transition function* $\delta : Q \times \Sigma \to L(Q)$ where $L(Q)$ is the set of positive FO sentences, called *transition specifications*, built on the vocabulary Q where each state $q \in Q$ is seen as a unary predicate, i.e. the least set of FO formulas containing formulas $q(x)$, $x = y$, $x \neq y$, and closed under conjunction, disjunction, existential and universal FO quantifications.

Remark that here counting means that the automaton is capable, via equality and inequality inside transition specifications, to count up to some bound the number of successors of vertices.

A tree-automaton $\mathcal{A}$ is called an *alternating modal tree-automaton* when, for each $q \in Q$, each $a \in \Sigma$, the FO formula $\delta(q, a)$ is built without the atomic formulas $x = y$ and $x \neq y$.

A tree-automaton $\mathcal{A}$ is called a *non deterministic counting tree-automaton* when, for each $q \in Q$, $a \in \Sigma$, $\delta(q, a)$ is a disjunction of formulas of the form

$$\exists x_1, \cdots, x_k \mathit{diff}(x_1, \cdots, x_k) \wedge q_{i_1}(x_1) \wedge \cdots \wedge q_{i_k}(x_k) \wedge \forall z, \mathit{diff}(z, x_1, \cdots, x_n) \Rightarrow \bigvee_{q' \in Q'} q'(z)$$

with any states $q_{i_1}, \ldots, q_{i_k}$ not necessarily distinct and any $Q' \subseteq Q$ where, again, *diff* predicates only says that each variable is distinct from any other.

Note that non derministic modal automata can also be defined (see [13]) but, apart for the non emptiness problem, they don't have all the interesting properties of usual notions of non deterministic automata such as, for instance, closure under projection. This comes from the fact the modal mu-calculus (or even polymodal logic) is not closed under set quantifiers as shown by the "formula" $\exists X(\Diamond X \wedge \Diamond \neg X)$.

Given a graph M, a *run* of $\mathcal{A}$ over M is a graph ρ which set of vertices V^ρ is some subset of the set of pairs $(s, q) \in S^M \times Q$ with $(r^M, q_0) \in V^\rho$ and which set of edges $E^\rho \subseteq V^\rho \times V^\rho$ is such that : for any pair $(s, q) \in V^\rho$, given the local structure $L^\rho_{s,q}$ over the vocabulary Q defined by $dom(L^\rho_{s,q}) = \{s' \in S^M : (s, s') \in R^M\}$ and, for each $p \in Q$, $p^{L^\rho_{s,q}} = \{s' : ((s, q), (s', p)) \in E^\rho\}$, one has

$$L^\rho_{s,q} \models \delta(q, \lambda(s))$$

A run ρ is called *functional* when, for any $s \in S^M$ there is at most one $q \in Q$ such that $(s, q) \in V^\rho$.

A run ρ of $\mathcal{A}$ over M is an *accepting run* when, for each infinite path π in ρ of the form $\pi = (r^M, q_0).(s_1, q_1).\cdots$ the minimum $\min\{\Omega(q_i) : |\{j \in I\!N : q_i = q_j\}| = \infty\}$ is even.

The next lemma shows that, although runs are defined over arbitrary graphs, these automata implicitly "read" trees as input.

Lemma 4.1 *For each graph M there is an accepting run of $\mathcal{A}$ over M iff there is an accepting run of $\mathcal{A}$ over $T(M)$.*

Proof. From left to right just notice that the unwinding of an accepting run of $\mathcal{A}$ over M is an accepting run of $\mathcal{A}$ over $T(M)$. The converse, less immediate, can be proven within parity game theory, the existence of an accepting run of $\mathcal{A}$ over M being equivalent to the existence of a memoryless winning strategy in some parity game built from $\mathcal{A}$ and M. □

For the next lemmas and theorems, we shall concentrate on trees.

Given an automaton $\mathcal{A}$, we denote by $L(\mathcal{A})$ the class of all trees M such that there exists an accepting run of $\mathcal{A}$ over M. The class $L(\mathcal{A})$ is called the language of trees recognized by $\mathcal{A}$.

The following theorem can be obtained from the results presented in [33]. It also follows from [12].

Theorem 4.2 *For each class of tree L, the following statements are equivalent :*
1. L is definable with an MSO sentence,
2. L is definable with a counting mu-calculus formula,
3. $L = L(\mathcal{A})$ for some alternating counting tree automaton $\mathcal{A}$,
4. $L = L(\mathcal{A})$ for some non deterministic[3] counting tree automaton $\mathcal{A}$.

and the next one follows from [32] and [14]

Theorem 4.3 *For each class of tree L, the following statements are equivalent :*
1. L is definable with a bisimulation invariant MSO sentence,
2. L is definable with a modal μ-calculus formula,
3. $L = L(\mathcal{A})$ for some modal tree automaton $\mathcal{A}$.

Some particular subclasses of tree-automaton that will be useful in the sequel. Automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \Omega, \delta \rangle$

[3] possibly with more parity indices

is called a ν-*automaton* (resp. $\nu\mu$-*automaton* or *Büchi automaton*) when $\Omega(Q) = \{0\}$ (resp. when $\Omega(Q) = \{0, 1\}$).

These automata characterize the ν-levels and $\nu\mu$-levels of the counting and modal mu-calculi in the following sense.

Lemma 4.4 (Expressiveness) *A class of tree L is recognized by a (counting or modal) ν-automaton (resp. $\nu\mu$-automaton) iff L is definable by a (modal or counting) mu-calculus formula of the ν-level (resp. of the $\nu\mu$-level).*

Proof. This lemma is a particular case of the well-known correspondence between level of the mu-calculus hierarchy and the number of parity indices needed in alternating tree-automata. This correspondance was first achieved, in the case of the binary tree, by Niwiński [24]. See [33] for a proof in the counting mu-calculus case. □

This implies in particular that the classes of languages recognized by ν-automata or $\nu\mu$-automata are closed under union and intersection.

For counting automata more properties are available :

Lemma 4.5 *The class of languages recognizable by counting ν-automata (resp. by counting $\nu\mu$-automata) is closed under projection.*

Proof. This lemma follows from the next two. □

Lemma 4.6 (Simulation) *A language recognized by a counting ν-automaton (resp. a counting $\nu\mu$-automaton) is also recognized by a non deterministic counting ν-automaton (resp. a non deterministic counting $\nu\mu$-automaton).*

Proof. Extension to arbitrary trees of (a part of) Muller and Schupp's simulation theorem [23] for alternating tree automata over the binary tree. □

and

Lemma 4.7 (Projection) *The projection of a language recognized by a non deterministic counting automaton is also recognized by a non deterministic automaton with the same set of states and parity function.*

Proof. When $\mathcal{A}$ is non deterministic counting one can restrict runs (over trees) to be functional without changing the language recognized by $\mathcal{A}$. Closure under projection immediately follows from this restriction. □

To conclude this section on automata, we recall here the heart of the bisimulation invariance result presented in [14] as the following lemma which will be used in the sequel :

Lemma 4.8 *For each non deterministic counting tree automaton $\mathcal{A}$ there exists a modal automaton $\mathcal{B}$, with the same set of states and parity function, such that, for each tree M, any infinite set κ, $T^\kappa(M) \in L(\mathcal{A})$ iff $M \in L(\mathcal{B})$.*

Proof. See [14] for a complete proof. The main idea is to define $\mathcal{B}$ as the automaton obtained from $\mathcal{A}$ by replacing all equalities or inequalities in the FO formula of δ by some true formula. □

5 Bisimulation invariance in monadic Σ_1

In this section, we prove theorem 1.1. For this, we first prove the analogue for unwinding invariance, from which, applying Lemma 4.4 and Lemma 4.8, we obtain the desired result.

So our goal is to prove the following theorem :

Theorem 5.1 *The unwinding invariant fragment of the level Σ_1 (resp. Π_1) in the monadic hierarchy equals the ν-level (resp. the μ-level) of the counting mu-calculus hierarchy.*

Proof. By duality, it is sufficient to prove the result for monadic Σ_1. Moreover, it is a classical result, from Lemma 4.4 stated above, that properties definable in the ν-level of the counting mu-calculus are definable in monadic Σ_1. So it remains to prove that :

Lemma 5.2 *Any unwinding invariant formula of monadic Σ_1 is equivalent to a formula of the the ν-level of the counting mu-calculus.*

In order to do so, one must understand that, as stated in the introduction, it is not sufficient to restrict our analysis to trees - although an unwinding invariant property is fully determined by its models among trees - because over trees, monadic Σ_1 is strictly more expressive than the ν-level of the counting mu-calculus as the (even FO) formula $\exists x p(x)$ shows.

First, remark that an unwinding invariant property only speaks about the vertices reachable from the root because any graph M has the same unwinding as the subgraphs induced by these vertices. This leads to the following definitions. Let $c(r_M)$ be the set of all vertices which are reachable from the root r_M via a (directed) path (called in the sequel the *directed connected component* induced by r_M). For each MSO sentence φ, let us define φ^c as the formula φ relativized to the directed connected component $c(r)$ of r, i.e. φ^c is obtain from φ replacing any first order or set quantification by quantifications over vertices or subsets of $c(r)$. With this definition and the previous remark it appears that if φ is invariant under unwinding then φ is equivalent to φ^c; in particular, if φ is in monadic Σ_1 then φ^c is also (definable) in monadic Σ_1.

So let φ be an unwinding invariant monadic Σ_1 formula. By the Gaifman normal form theorem for first order logic, there is some integer k such that φ is of the form

$$\varphi = \exists \vec{Z}.\varphi_1$$

with $\vec{Z}$ a finite vector of sets variables and φ_1 is a finite boolean combination of FO formulas $G(\vec{Z})$ of the form

$$G(\vec{Z}) = \exists u_1, \ldots, u_l.\theta(u_1, \ldots, u_l, \vec{Z})$$

where $\theta(u_1, \ldots, u_l, \vec{Z}, \vec{Y})$ is a formula stating that for all distinct indices s and t among $[1, l]$, $dist(u_s, u_t) > 2k$ and $Ball(u_s, k) \models \psi_s(\vec{Z})$ for some FO formulas $\psi_s(\boldsymbol{Z})$, with $dist(x, y)$ defined as the length of the shortest *undirected* path from x to y and $Ball(x, k)$ is defined as the substructure of M induced by the set of all vertices y such that $dist(x, y) \leq k$.

For notational simplicity we assume that φ is of the form

$$\varphi = \exists \vec{Z}.G(\vec{Z}) \wedge \neg G'(\vec{Z})$$

with $G(\vec{Z})$ of the form $\exists u\theta(u, \vec{Z})$ and $G'(\vec{Z})$ of the form $\exists u'\theta'(u', \vec{Z})$. One can check that this proof easily extends to the general case.

The relativization φ^c of φ to the strongly connected components of r is then given by :

$$\varphi^c = \exists \vec{Z}.G^c(\vec{Z}) \wedge \neg G'^c(\vec{Z})$$

with $G^c(\vec{Z})$ given by $\exists u \in c(r).\theta^c(u, \vec{Z})$ and $G'^c(\vec{Z})$ given by $\exists u' \in c(r).\theta'^c(u', \vec{Z})$.

Now, we know that the formula φ^c cannot have for arbitrarily large integers n a model M_n, where the points of $c(r)$ satisfying θ^c have (directed) distance more than n from r. Otherwise, the ultraproduct of the M_ns modulo any non principal ultrafilter, would not satisfy φ^c, contrary to the Σ_1 definability of φ^c and Łos ultraproduct theorem (see for instance [29]) which says that the class of models of any Σ_1 formula is closed under ultraproduct.

So given integer $\overline{n}$ such that no model M_n for $n > \overline{n}$ satisfies φ^c, it turns out that formula φ^c is equivalent to formula $\exists \vec{Z}.\neg G'^c(\vec{Z}) \wedge G^{\overline{n}}(\vec{Z})$ with

$$G^{\overline{n}}(\vec{Z}) = \exists u \in c_{\overline{n}}(r).\theta^c(u, \vec{Z})$$

and $c_{\overline{n}}(r)$ the set of all points directly accessible from r in at most $\overline{n}$ steps.

Now it is not difficult to see that $\neg G'^c(\vec{Z})$ is a fixpoint formula of the ν-level over trees (i.e. unwindings) and $G^{\overline{n}}(\vec{Z})$ is even a fixpoint free formula on unwindings as well. By unwinding invariance, this says that φ is equivalent to some formula of the form $\exists \vec{Z}\varphi_{\alpha'}(\vec{Z})$ with $\alpha' \in N_1$.

Then, over trees, Lemma 4.5, ensures $\exists \vec{Z}\varphi_{\alpha'}(\vec{Z})$ is equivalent to some φ_α for some α in the ν-level as well hence, again by invariance under unwinding, φ is equivalent over arbitrary models to φ_α. □

6 Bisimulation invariance in monadic Σ_2

In this section, we prove theorem 1.2. For this, again, we first prove the analogue for unwinding invariance, from which, applying Lemma 4.4 and Lemma 4.8 we obtain the desired result. So our goal is to prove the following theorem :

Theorem 6.1 *The unwinding invariant fragment of the level Σ_2 (resp. Π_2) in the monadic hierarchy equals the $\nu\mu$-level (resp. the $\mu\nu$-level) of the counting mu-calculus hierarchy.*

Proof. By duality, it is again sufficient to prove the result for monadic Σ_2. Moreover, it is again a classical result, from Lemma 4.4, that properties definable in the $\nu\mu$-level of the counting mu-calculus are definable in monadic Σ_2. So it remains to prove that :

Lemma 6.2 *Any unwinding invariant formula of monadic Σ_2 is equivalent to a formula of the the $\nu\mu$-level of the counting mu-calculus.*

Proof. Somehow, the proof in the case of Σ_2 is simpler than Σ_1 for it is true that, over trees, any monadic Σ_2 formula is equivalent to a $\nu\mu$-formula which remains to be shown.

For this, we use definability in weak monadic second order logic as an intermediate step. Remember that weak monadic second order logic is monadic second order logic with set quantification restricted to finite sets.

A priori, using weak MSOL doesn't make sense. Indeed, over arbitrary trees, weak MSOL is incomparable with MSOL. However, Theorem 4.2 and the definition of tree automata show that analyzing MSOL over trees can be made over finitely branching trees only. In fact any MS formula satisfiable over the class of trees has a model which is finitely branching, i.e. with finitely many successors from each vertex.

For this reason, we can restrict our study to finitely branching trees and then weak MSOL is a fragment of MSOL since, in this case, finite sets are definable in MSOL.

The sketch of the proof is then the following. First we prove

Lemma 6.3 *Any language of (finitely branching) trees definable in monadic Σ_1 is definable in weak MSOL.*

Then, by closure of weak MSOL under negation, this shows that monadic Π_1 is also included into weak MSOL. Hence monadic Σ_2 is included into the existential projection of weak MSOL. Now, because the class of languages definable by $\nu\mu$-automaton is closed under projection (see Lemma 4.5) we prove

Lemma 6.4 *Any languages of (finitely branching) trees definable in weak MSOL is recognizable by a $\nu\mu$-automaton.*

which conclude the proof of Lemma 6.2. □

In order to prove Lemma 6.3 we can adapt the work of Lenzi [19], to the case of finitely branching trees. Another

approach, following the idea of Skurczinski [31], is to use weak $\nu\mu$-automaton as an intermediate step.

We recall here that a tree automaton $\mathcal{A}$ is a *weak automaton* when, for any $q \in Q$, any $a \in \Sigma$, for each states q' occurring in formula $\delta(q,a)$, $\Omega(q) \leq \Omega(q')$.

Then, adapting the proof presented in [30] for the k-ary case, one has :

Lemma 6.5 *Any FO definable tree languages is recognizable by a weak strongly non deterministic $\nu\mu$-automaton.*

But then, since languages recognizable by strongly non deterministic weak $\nu\mu$-automaton are closed under projection, it is sufficient to show that

Lemma 6.6 *Any languages of (finitely branchinbg trees) recognizable by a weak automaton is definable by a weak MSOL formula.*

And this last lemma is an adaptation of similar result, by Mostowski [22], over the binary tree. □

For Lemma 6.4, it shall be clear that it can be proved extending, in a quite straightforward way an analogous proof due to Rabin [27] in the binary case.

This concludes the proof of Theorem 6.1 for, applying Lemma 4.4, languages recognizable by $\nu\mu$-automata equal languages definable by (counting) fixpoint formulas of the $\nu\mu$-level. □

7 Above the level Σ_2

In this section, we prove Theorem 1.3 and Theorem 1.4. For this, we assume that the reader has a general knowledge of the theory of parity games[4]. If not, Jurdziński's [16] gives an appropriate, and up to date, overview of the topic.

From [4] we know that, given an integer k, expressing the fact that a position in an arbitrary parity game with sets of parity indices $[0,k]$ cannot be done with any mu-calculus formula of the level N_k. From [2] we know that this is still the case restricted to games of degree two.

Remark that in monadic second order logic, this may also be difficult to express because in some sense it requires some, at least implicit, construction of a (memoryless) strategy for player 0 which is winning for any plays starting in the distinguished position. And winning strategies are peculiar sets of edges which are, in general, not even definable in MSOL.

Still we prove Theorem 1.3 redefining binary games on graphs (over a more complex signature) on which guessing a winning strategy will become possible with a single existential set quantification. The main difficulty is only to ensure that such a definition leads to bisimulation invariant class of parity games.

More precisely, given some integer $k > 2$, given *Prop* defined by $Prop = \{p_l, p_r, p_0, \cdots, p_k\}$, any graph M such that both $\{p_l^M, p_r^M\}$ and $\{p_0^M, \cdots, p_k^M\}$ are partitions of the set of vertices S^M reachable from the source r - which is a bisimulation invariant property - is from now on interpreted as a parity game as follows :

1. any position (reachable from the root) is a position of player 0,
2. a move from such a position is made as follows : player 0 chooses one predicate $p_x \in \{p_l, p_r\}$ and then player 1 chooses the new position $y \in S^M$ such that $y \in p^M(y)$ and $(x,y) \in R^M$,
3. disjoint predicates $p_0, \ldots, p_k$ encode the parity indices of each of these positions.

Theorem 1.3 is then a consequence of the following lemma :

Lemma 7.1 *For each integer $k > 2$, the class W_0^k of (encoded) games over the set of indices $[0,k]$ where the root is a winning position for player 0 is bisimulation closed, definable with a monadic Σ_3 formula and not definable in the level N_k of the mu-calculus.*

Proof. First observe that any bisimulation relation relates winning positions for player 0 to winning position for player 0 so the class W_0^k is indeed bisimulation closed.

Then, it is clear that any binary game can be encoded in such a way. Moreover, computing with a mu-calculus formula the fact that the root r is a winning positions for player 0 in this encoding is as difficult - in terms of number of alternations of least and greatest fixpoints - as computing the fact that the root r is a winning position for the same player in binary games so, following the result of Arnold [2], it requires at least $k+1$ alternations of least and greatest fixpoints.

Now, to conclude the proof it is sufficient to show that the class W_0^k is definable in monadic Σ_3. But this can easily be achieved as follows : first, with some existential set quantifier, one can guess a winning strategy for player 0, e.g. guessing the set of positions X from which player 0 chooses predicate p_r. Then it is clear that a $\mu\nu$-formula of the mu-calculus (henceforth a monadic Π_2 formula) is sufficient to check that this set X is indeed a winning strategy for player 0 in any plays that start at the root. Indeed, one has to check the minimal parity condition on any cycle reachable from the root when player 0 follows the strategy given by set X. In the intended $\mu\nu$-formula, one least fixpoint enables us to reach any of these cycles and then, one nested greatest fixpoint enables us to check that the minimum parity index met on each of these cycles is even.

[4] with the winning criteria defined as an even minimal index met infinitely often... !

Guessing a winning strategy and checking that it is winning for player 0 can thus be expressed in monadic Σ_3. □

The proof of Theorem 1.4 is also almost done. Indeed, from the proof of previous lemmas it is clear that *with one existential quantification over sets of edges* the winning position for player 0 can be expressed as a monadic Σ_3 unary predicate. But it also follows from Lemma 4.4 that checking a fixpoint formula on a graph can be done via a monadic Σ_1 transduction which leads to computing winning positions with as many parity indices as the alternation depth of the formula. Moreover, if the input graph is of bounded degree (or bounded tree-width) then the resulting parity game is also of bounded degree (or bounded tree-with). Now Courcelle shows that over graphs with bounded degree (or tree-width) quantification over edges can be "simulated" by quantifications over vertices via, again, a monadic Σ_1 transduction. Altogether, this says that over graphs of bounded degree (or bounded tree-width) mu-calculus formulas can be translated into monadic Σ_3 formulas. This concludes the proof of Theorem 1.4. □

References

[1] M. Ajtai and R. Fagin. Reachability is harder for directed rather than undirected finite graphs. *Journal of Symbolic Logic*, 55:113–150, 1990.

[2] A. Arnold. The mu-calculus alternation-depth hierarchy over the binary tree is strict. *Theoretical Informatics and Applications*, 33:329–339, 1999.

[3] J. Benthem. *Modal Correspondance Theory*. PhD thesis, University of Amsterdam, 1976.

[4] J. Bradfield. The modal mu-calculus alternation hierarchy is strict. *Theoretical Comp. Science*, 195:133–153, 1998.

[5] B. Courcelle. The monadic second-order logic of graphs VI: On several representations of graphs by logical structures. *Discrete Applied Mathematics*, 54:117–149, 1994.

[6] E. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theor. Comp. Science (vol. B)*, pages 995–1072. Elsevier, 1990.

[7] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*, volume 7. SIAM-AMS, 1974.

[8] S. Ghilardi and M. Zawadowski. A sheaf representation and duality for finitely presented Heyting algebras. *Journal of Symbolc Logic*, 60:911–939, 1995.

[9] E. Grädel, C. Hirsch, and M. Otto. Back and Forth Between Guarded and Modal Logics. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science LICS 2000*, pages 217–228, 2000.

[10] T. Hafer and W. Thomas. Computationnal tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *ICALP'87*, pages 269–279. LNCS 267, 1987.

[11] M. Hollenberg. *Logic and Bisimulation*. PhD thesis, Utrecht University, 1998.

[12] D. Janin. *Propriétés logiques du non déterminisme et mu-calcul modal*. PhD thesis, Université de Bordeaux I, 1996.

[13] D. Janin and I. Walukiewicz. Automata for the modal mu-calculus and related results. In *Math. Found of Comp. Science*. LNCS 969, 1995.

[14] D. Janin and I. Walukiewicz. On the expressive completeness of the modal mu-calculus w.r.t. monadic second order logic. In *CONCUR'96*, pages 263–277. LNCS 1119, 1996.

[15] M. Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[16] M. Jurdziński. Small progress measures for solving parity games. In *Symp. on Theor. Aspects of Computer Science*, pages 290–301. LNCS 1770, 2000.

[17] D. Kozen. Results on the propositional μ-calculus. *Theoretical Comp. Science*, 27:333–354, 1983.

[18] G. Lenzi. *The Mu-calculus and the Hierarchy Problem*. PhD thesis, Scuola Normale Superiore, Pisa, 1997.

[19] G. Lenzi. A new logical characterization of Büchi automata. In *Symp. on Theor. Aspects of Computer Science*, 2001.

[20] O. Matz and W. Thomas. The monadic quantifier alternation hierarchy over finite graphs is infinite. In *IEEE Symp. on Logic in Computer Science*, pages 236–244, 1997.

[21] F. Moller and A. Rabinovich. On the expressive power of CTL*. In *IEEE Symp. on Logic in Computer Science*, pages 360–369, 1999.

[22] A. Mostowski. Hierarchies of weak automata on weak monadic formulas. *Theoretical Comp. Science*, 83:323–335, 1991.

[23] D. E. Muller and P. E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Comp. Science*, 141:67–107, 1995.

[24] D. Niwiński. On fixed point clones. In *13th ICALP*, pages 464–473, 1986.

[25] M. Otto. Bisimulation-invariant Ptime and higher-dimensional mu-calculus. *Theoretical Comp. Science*, 224:237–265, 1999.

[26] M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[27] M. O. Rabin. Weakly definable relations and special automata. In *Mathematical Logic and Foundation of Set Theory*, pages 1–23. North Holland, 1970.

[28] E. Rosen. Modal logic over finite structures. *Journal of Logic, Language and Information*, 6:427–439, 1997.

[29] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Mass., 1967.

[30] J. Skurczyński. On three hierarchies of weak SkS formulas. Aachener Informatik-Berichte 90-3, RWTH Aachen, 1990.

[31] J. Skurczyński. A characterization of Büchi tree automata. unpublished manuscript, Gdansk University, 2000.

[32] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81:249–264, 1989.

[33] I. Walukiewicz. Monadic second order logic on tree-like structures. In *Symp. on Theor. Aspects of Computer Science*, 1996. LNCS 1046.

Focus Games for Satisfiability and Completeness of Temporal Logic

Martin Lange Colin Stirling
LFCS, Division of Informatics, University of Edinburgh,
JCMB, King's Buildings, Edinburgh, EH9 3JZ
{martin,cps}@dcs.ed.ac.uk

Abstract

We introduce a simple game theoretic approach to satisfiability checking of temporal logic, for LTL and CTL, which has the same complexity as using automata. The mechanisms involved are both explicit and transparent, and underpin a novel approach to developing complete axiom systems for temporal logic. The axiom systems are naturally factored into what happens locally and what happens in the limit. The completeness proofs utilise the game theoretic construction for satisfiability: if a finite set of formulas is consistent then there is a winning strategy (and therefore construction of an explicit model is avoided).

1 Introduction

The automata theoretic approach to satisfiability checking for temporal logic is very popular and successful [6, 17]. However there is a cost with the involvement of automata mechanisms and in particular the book keeping implicit in the product construction, when a local automaton is paired with an eventuality automaton. While this is not an impediment for checking satisfiability it appears to be for other formal tasks such as showing that an axiomatisation of a temporal logic is complete. When proving completeness, one needs to establish that a finite consistent set of formulas is satisfiable. It is not known, in general, how to plug into such a proof automata theoretic constructions (such as product and determinisation) for satisfiability. Instead standard completeness proofs either appeal to "canonical" structures built from maximal consistent sets [15, 8] or tableaux which explicitly build models from consistent sets, as illustrated by the delicate proofs of completeness for CTL* [14] and modal μ-calculus [18], and even the proofs of completeness for LTL [7, 13] (future linear time logic) and CTL [5] (computation tree logic).

In this paper we introduce a simple game theoretic approach to satisfiability checking of temporal logic, for LTL and CTL, which has the same complexity as using automata. The mechanism involved, the use of a "focus", is both explicit and transparent, and underpins a novel approach to developing complete axiom systems for temporal logic. The axiom systems are naturally factored into what happens locally and what happens in the limit. The completeness proofs use the game theoretic construction for satisfiability: if a finite set of formulas is consistent then there is a winning strategy (and therefore construction of an explicit model is avoided).

Although the origin of these games is model checking CTL* [12], it remains to be seen if the game technique extends to satisfiability checking of CTL* and modal μ-calculus. Moreover, it remains to be seen if the technique is practically viable for testing satisfiability of LTL and CTL.

2 LTL

We present LTL [7] in positive form, where only atomic formulas are negated. Let Prop be a family of atomic propositions closed under negation, where $\neg\neg q = q$, and containing the constants $\mathtt{tt}$ (true) and $\mathtt{ff}$ (false). Formulas of LTL are built from Prop using boolean connectives $\vee$ and $\wedge$, the unary temporal operator X (next) and the binary temporal connectives U (until) and its dual R (release).

We assume a usual ω-model for formulas, consisting of an infinite sequence of states which are maximal consistent sets of atomic formulas. A state s therefore obeys the condition that for any $q \in \mathsf{Prop}$, $q \in s$ iff $\neg q \notin s$, and $\mathtt{tt} \in s$ and $\mathtt{ff} \notin s$. The semantics inductively defines when an ω-sequence of states σ satisfies a formula Φ, written $\sigma \models \Phi$. In the case of $q \in \mathsf{Prop}$, $\sigma \models q$ iff q is in the initial state of σ. The clauses for the boolean connectives are as usual. If $\sigma = s_0 s_1 \ldots$ and $i \geq 0$ then $\sigma^i = s_i s_{i+1} \ldots$ is the ith suffix of σ. The remaining clauses are as follows.

$$\begin{array}{lll} \sigma \models X\Phi & \text{iff} & \sigma^1 \models \Phi \\ \sigma \models \Phi U \Psi & \text{iff} & \exists i \geq 0.\, \sigma^i \models \Psi \text{ and} \\ & & \quad \forall j : 0 \leq j < i.\, \sigma^j \models \Phi \\ \sigma \models \Phi R \Psi & \text{iff} & \forall i \geq 0.\, \sigma^i \models \Psi \text{ or} \\ & & \quad \exists j : 0 \leq j < i.\, \sigma^j \models \Phi \end{array}$$

0-7695-1281-X/01 $10.00 © 2001 IEEE

We assume that $F\Psi$ (eventually Ψ) abbreviates $\mathtt{tt}U\Psi$ and its dual $G\Psi$ (always Ψ) abbreviates $\mathtt{ff}R\Psi$. The meanings of U and R are determined by their fixed point definitions, $\Phi U\Psi$ is the least solution to $\alpha = \Psi \vee (\Phi \wedge X\alpha)$ whereas $\Phi R\Psi$ is the largest solution of $\alpha = \Psi \wedge (\Phi \vee X\alpha)$.

A formula Φ is satisfiable if there is a model σ such that $\sigma \models \Phi$. In the naive tableau approach to deciding satisfiability, one constructs an "or" decision tree. The root is a finite set of initial formulas, and the decision question is whether their conjunction is satisfiable. Child nodes are produced by local rules on formulas. A node $\Gamma \cup \{\Phi \wedge \Psi\}$ has child $\Gamma \cup \{\Phi, \Psi\}$. A node $\Gamma \cup \{\Phi \vee \Psi\}$ has two children $\Gamma \cup \{\Phi\}$ and $\Gamma \cup \{\Psi\}$. Formulas $\Phi U\Psi$ and $\Phi R\Psi$ are replaced by their fixed point unfolding, $\Psi \vee (\Phi \wedge X(\Phi U\Psi))$ and $\Psi \wedge (\Phi \vee X(\Phi R\Psi))$. After repeated applications of these rules, a node without children has the form $\{q_1, \ldots, q_n, X\Phi_1, \ldots, X\Phi_m\}$, where each $q_i \in \mathsf{Prop}$. If the set $P = \{q_1, \ldots, q_n\}$ is unsatisfiable then the node is an unsuccessful leaf. If P is satisfiable and $m = 0$ then the node is a successful leaf. Otherwise a new child $\{\Phi_1, \ldots, \Phi_m\}$ is produced, which amounts to moving to a new state.

Nodes with until or release formulas may continually produce children, and therefore one also needs another criterion for when a node counts as a leaf. An obvious candidate is when a node is a repetition, contains the same formulas as an earlier node (and in between there is at least one application of the new state rule). Whether or not such a leaf is successful will depend on whether formulas are the result of the fixed point unfolding of a release or an until formula. A repeat of $\Phi R\Psi$ should be successful whereas a repeat of $\Phi U\Psi$ is unsuccessful.

Consider the following example decision tree, where set braces are dropped (and $\mathtt{tt}$ and $\mathtt{ff}$ are dispensed with and so the unfolding of $F\Psi$ is $\Psi \vee XF\Psi$ and the unfolding of $G\Psi$ is $\Psi \wedge XG\Psi$).

$$\dfrac{\dfrac{\dfrac{Fq, XGFq}{q \vee XFq, XGFq}}{\dfrac{\dfrac{q, XGFq}{GFq}\,\text{Next}}{Fq \wedge XGFq}} \qquad \dfrac{\dfrac{XFq, XGFq}{Fq, GFq}\,\text{Next}}{Fq, Fq \wedge XGFq}}{Fq, XGFq} \qquad Fq, XGFq$$

Next labels a transition to a new state. Both leaves in this tree are repetitions of the root. However the left leaf should count as successful because the formula Fq at the initial node is "fulfilled" in the left branch, giving the model s_0^ω where $q \in s_0$. In contrast Fq is not fulfilled in the right branch and is thereby "regenerated", and therefore the right leaf should count as unsuccesful.

The problem of which fixed points are regenerated disappears in the automata theoretic approach to satisfiability [17]. Roughly speaking, the decision tree is then only part of the story. It is captured by the "local" automaton and one also needs to factor in the "eventuality" automaton which automatically deals with regeneration of fixed points, and therefore the problem does not arise. However the cost is the use of the product construction between the two automata. While this is not an impediment for checking satisfiability it appears to be for other formal tasks such as showing that an axiomatisation of a temporal logic is complete.

We now show that a simple game theoretic approach to satisfiability checking, where the mechanisms are both explicit and transparent, has the virtue that it also leads to very simple proofs of completeness for both LTL and CTL.

3 Games for LTL

In the naive tableau approach to satisfiability there are "or" choices but there are no "and" choices. Recasting as a game, "or" choices are $\exists$-choices for the player $\exists$ and "and" choices are $\forall$-choices for the player $\forall$. The role of player $\exists$ is that of verifier, "I want to show that the initial set of formulas is satisfiable" whereas the role of $\forall$ is that of refuter, "I want to show that the initial set of formulas is unsatisfiable". In a position $\Gamma, \Phi_1 \vee \Phi_2$ player $\exists$ chooses the disjunct Φ_i, and play continues from the position Γ, Φ_i. The idea is that $\exists$ ($\forall$) has a winning strategy iff the initial set of formulas is satisfiable (unsatisfiable).

We need to force player $\forall$ to make choices. A new component, the "focus", is introduced into a set of formulas for this purpose. One of the formulas in a position is in focus. We write $[\Phi].\Gamma$ to represent the position $\Gamma \cup \{\Phi\}$ when Φ is in focus. Player $\forall$ chooses which formula is in focus. If it is an "and" formula then $\forall$ chooses which subformula to keep in focus. During a play $\forall$ may also change mind, and move the focus to a different formula.

Given a starting formula Φ_0 (the conjunction of the initial formulas) we will define its focus game $\mathsf{G}(\Phi_0)$. The set of subformulas of Φ_0, $\mathrm{Sub}(\Phi_0)$, is defined as expected but with the requirement that the unfolding of an until $\Psi \vee (\Phi \wedge X(\Phi U\Psi))$ is a subformula of $\Phi U\Psi$ and the unfolding of a release $\Psi \wedge (\Phi \vee X(\Phi R\Psi))$ is a subformula of $\Phi R\Psi$. A position in a play of $\mathsf{G}(\Phi_0)$ is an element $[\Phi], \Gamma$ where $\Phi \in \mathrm{Sub}(\Phi_0)$ and $\Gamma \subseteq \mathrm{Sub}(\Phi_0) - \{\Phi\}$. A play of the game $\mathsf{G}(\Phi_0)$ is a sequence of positions $P_0 P_1 \ldots P_n$ where P_0 is the initial position $[\Phi_0]$, and the change in position P_i to P_{i+1} is determined by one of the moves of Figure 1. They are divided into three groups. First are rules for $\exists$ who chooses disjuncts in and out of focus. Second are the moves for player $\forall$ who chooses which conjunct remains in focus and who also can change focus with the rule change. Finally, there are the remaining moves which do not involve

Player $\exists$

$$\frac{[\Phi_0 \vee \Phi_1], \Gamma}{[\Phi_i], \Gamma} \qquad \frac{[\Phi], \Phi_0 \vee \Phi_1, \Gamma}{[\Phi], \Phi_i, \Gamma}$$

Player $\forall$

$$\frac{[\Phi_0 \wedge \Phi_1], \Gamma}{[\Phi_i], \Phi_{1-i}, \Gamma} \qquad \frac{[\Phi], \Psi, \Gamma}{[\Psi], \Phi, \Gamma} \text{ change}$$

Other moves

$$\frac{[\Phi U \Psi], \Gamma}{[\Psi \vee (\Phi \wedge X(\Phi U \Psi))], \Gamma} \qquad \frac{[\Phi'], \Phi U \Psi, \Gamma}{[\Phi'], \Psi \vee (\Phi \wedge X(\Phi U \Psi)), \Gamma}$$

$$\frac{[\Phi R \Psi], \Gamma}{[\Psi \wedge (\Phi \vee X(\Phi R \Psi))], \Gamma} \qquad \frac{[\Phi'], \Phi R \Psi, \Gamma}{[\Phi'], \Psi \wedge (\Phi \vee X(\Phi R \Psi)), \Gamma}$$

$$\frac{[\Phi], \Phi_0 \wedge \Phi_1, \Gamma}{[\Phi], \Phi_0, \Phi_1, \Gamma} \qquad \frac{[X\Phi_1], \ldots, X\Phi_m, q_1, \ldots, q_n}{[\Phi_1], \ldots, \Phi_m} \text{ next}$$

Figure 1. Game moves

any choices, and so neither player is responsible for them. These include the fixed point unfolding of until and release in and out of focus, the removal of $\wedge$ out of focus and the next state rule, next, where the focus remains with the subformula of the next formula in focus. It is therefore incumbent on $\forall$ to make sure that an X formula is in focus when next is applied.

The next ingredient in the definition of the game is the winning conditions for a player, when a play counts as a win.

Definition 1 Player $\forall$ wins the play $P_0, \ldots, P_n$ if

1. P_n is $[q], \Gamma$ and (q is `ff` or $\neg q \in \Gamma$) or
2. P_n is $[\Phi U \Psi], \Gamma$ and for some $i < n$ the position P_i is $[\Phi U \Psi], \Gamma$ and between $P_i \ldots P_n$ player $\forall$ has not applied the rule change.

Therefore $\forall$ wins if there is a simple contradiction or a repeat position with the same until formula in focus and no application of change between the repeats.

Definition 2 Player $\exists$ wins the play $P_0, \ldots, P_n$ if

1. P_n is $[q_1], \ldots, q_n$ and $\{q_1, \ldots, q_n\}$ is satisfiable or
2. P_n is $[\Phi R \Psi], \Gamma$ and for some $i < n$ the position P_i is $[\Phi R \Psi], \Gamma$ or
3. P_n is $[\Phi], \Gamma$ and for some $i < n$ the position P_i is $[\Phi], \Gamma$ and between $P_i \ldots P_n$ player $\forall$ has applied the rule change.

So $\exists$ wins if player $\forall$ is unable to focus on a X formula so that next can be applied when the atomic formulas are satisfiable. The other two conditions cover repeat positions. First is the case if the repeat position has the same release formula in focus, and second is the case of a repeat when the same formula is in focus and change has been applied between the repeat positions. The following upper bound on the length of a play is obvious.

Fact 1 *Every play of* $\mathsf{G}(\Phi_0)$ *has finite length less than* $|\mathrm{Sub}(\Phi_0)| \times 2^{|\mathrm{Sub}(\Phi_0)|}$.

A player wins the game $\mathsf{G}(\Phi_0)$ if the player is able to win every play of the game, that is has a winning strategy[1]. The following is a simple consequence of Fact 1 and the fact that the winning conditions are mutually exclusive.

Fact 2 *Every game* $\mathsf{G}(\Phi_0)$ *has a unique winner.*

Next we come to the game characterisation of satisfiability, which we split into two halves.

Proposition 1 *If* $\exists$ *wins the game* $\mathsf{G}(\Phi_0)$ *then* Φ_0 *is satisfiable.*

Proof: Assume $\exists$ wins the game $\mathsf{G}(\Phi_0)$. Consider the play where $\forall$ uses the following optimal strategy. Let $\Phi_1 U \Psi_1 \ldots, \Phi_n U \Psi_n$ be a priority list of all until subformulas of Φ_0, in decreasing order of size. We say that $\Phi U \Psi$ is present in a position P if either $\Phi U \Psi \in P$ or $\Psi \vee (\Phi \wedge X(\Phi U \Psi)) \in P$ or $X(\Phi U \Psi) \in P$. Player $\forall$ starts with the focus on Φ_0. If the formula in focus is a release formula $\Phi R \Psi$ and Ψ contains an until subformula then $\forall$ chooses Ψ when the release formula is unfolded. If the formula is a conjunction then $\forall$ chooses a conjunct with an until subformula. If the focus remains on a release formula or ends up on a member of Prop then $\forall$ changes focus, if this is possible, to the until formula which is present in the position and which is earliest in the priority list. If the focus is on an until formula $\Phi_i U \Psi_i$ then $\forall$ keeps the focus on it until it is "fulfilled", that is until player $\exists$ chooses Ψ_i when it is unfolded. This until formula is then moved to the end of the priority list. Player $\forall$ then changes focus to the earliest until formula in the priority list which is present in the position, if this is possible. This argument is then repeated. By assumption player $\exists$ wins against this strategy, and the play has finite length. It is now straightforward to extract an eventually cyclic model from the play, where every until formula present in some position will be fulfilled. □

Next we prove the converse of Proposition 1. One proof is to show how a winning strategy for $\exists$ can be extracted

[1] Formally a winning strategy, see for example [9], for player $\exists$ is a set of rules π of the form, if the play so far is $P_0 \ldots P_n$ and P_n is $[\Phi_0 \vee \Phi_1], \Gamma$ ($[\Phi], \Phi_0 \vee \Phi_1, \Gamma$) then choose $[\Phi_i], \Gamma$ ($[\Phi], \Phi_i, \Gamma$). Similarly for player $\forall$. A play obeys π if all the moves played by the player obey the rules in π. A strategy π is winning for a player if she wins every play in which she uses π.

from a model of Φ_0. However we provide an alternative proof which is the key to obtaining a complete axiom system. We utilise an observation from fixed point logics about least fixed points. Given Park's fixed point induction principle (1) below and that a fixed point is equivalent to its unfolding (2), Lemma 1 below holds (as observed by a number of researchers, for instance [10, 15, 19]). Standard substitution is assumed, $\Psi\{\Phi/Y\}$ is the replacement of all free occurrences of Y in Ψ with Φ. Moreover we write $\models \Phi$ to mean Φ is valid (true everywhere in all models).

(1) if $\models \Psi\{\Phi/Y\} \rightarrow \Phi$ then $\models \mu Y.\Psi \rightarrow \Phi$
(2) $\models \mu Y.\Psi \leftrightarrow \Psi\{\mu Y.\Psi/Y\}$

Lemma 1 *If Y is not free in Φ and $\Phi \wedge \mu Y.\Psi$ is satisfiable then the formula $\Phi \wedge \Psi\{(\mu Y.\neg\Phi \wedge \Psi)/Y\}$ is satisfiable.*

Proof: Suppose $\Phi \wedge \mu Y.\Psi$ is satisfiable, but $\models \Psi\{(\mu Y.\neg\Phi \wedge \Psi)/Y\} \rightarrow \neg\Phi$. Therefore $\models \Psi\{(\mu Y.\neg\Phi \wedge \Psi)/Y\} \rightarrow \neg\Phi \wedge \Psi\{(\mu Y.\neg\Phi \wedge \Psi)/Y\}$. Hence by (2) $\models \Psi\{(\mu Y.\neg\Phi \wedge \Psi)/Y\} \rightarrow \mu Y.\neg\Phi \wedge \Psi$ and so by (1) $\models \mu Y.\Psi \rightarrow \neg\Phi$ which contradicts that $\Phi \wedge \mu Y.\Psi$ is satisfiable. □

Lemma 1 sanctions the following property of until unfolding.

Lemma 2 *If $\Phi' \wedge (\Phi U \Psi)$ is satisfiable then $\Phi' \wedge (\Psi \vee (\Phi \wedge X((\Phi \wedge \neg\Phi')U(\Psi \wedge \neg\Phi'))))$ is satisfiable.*

Proof: Assume $\Phi' \wedge (\Phi U \Psi)$ is satisfiable. So there is a model σ such that $\sigma \models \Phi'$ and $\sigma \models \Phi U \Psi$, and therefore $\sigma^i \models \Psi$ and $\sigma^j \models \Phi$ for $j : 0 \leq j < i$, for some $i \geq 0$. Also assume $\Phi' \wedge (\Psi \vee (\Phi \wedge X((\Phi \wedge \neg\Phi')U(\Psi \wedge \neg\Phi'))))$ is not satisfiable, and so the following validity holds $\models \Phi' \rightarrow (\neg\Psi \wedge (\neg\Phi \vee X((\neg\Phi \vee \Phi')R(\neg\Psi \vee \Phi'))))$. Because $\sigma \models \Phi'$ therefore $\sigma \models \neg\Psi \wedge (\neg\Phi \vee X((\neg\Phi \vee \Phi')R(\neg\Psi \vee \Phi')))$. So $\sigma \models \neg\Psi$ and because $\sigma \models \Phi U \Psi$ it follows that $\sigma \models \Phi$. And so $\sigma \models X((\neg\Phi \vee \Phi')R(\neg\Psi \vee \Phi'))$, and therefore $\sigma^1 \models (\neg\Phi \vee \Phi')R(\neg\Psi \vee \Phi')$. And so $\sigma^1 \models \neg\Psi \vee \Phi'$ and $\sigma^1 \models \neg\Phi \vee \Phi' \vee X((\neg\Phi \vee \Phi')R(\neg\Psi \vee \Phi'))$. If $\sigma^1 \models \Phi'$ then $\sigma^1 \models \neg\Psi$ by the valid formula above, and so $\sigma^1 \models \neg\Psi$ and because $\sigma^1 \models \Phi U \Psi$ it follows that $\sigma^1 \models \Phi$, and so $\sigma^1 \models X((\neg\Phi \vee \Phi')R(\neg\Psi \vee \Phi'))$. The argument is now repeated for subsequent σ^j, $j \geq 0$, which contradicts that $\sigma \models \Phi U \Psi$. □

Proposition 2 *If Φ_0 is satisfiable then player $\exists$ wins the game $\mathsf{G}(\Phi_0)$.*

Proof: Assume that Φ_0 is satisfiable. We show that player $\exists$ wins the game $\mathsf{G}(\Phi_0)$. The idea is that $\exists$ always chooses a move which preserves satisfiability (and $\forall$ has to choose moves which preserve satisfiability). If $\Gamma \wedge (\Phi_0 \vee \Phi_1)$ is satisfiable then $\Gamma \wedge \Phi_i$ is satisfiable for at least one $i \in \{0, 1\}$, and so player $\exists$ chooses such an i. If the position is $[\Phi U \Psi], \Gamma$ where the until formula is in focus then player $\exists$ adorns the interpretation of it when it is unfolded, $[\Psi \vee (\Phi \wedge X(\Phi_{\neg\Gamma} U \Psi_{\neg\Gamma}))], \Gamma$ where $\Phi_{\neg\Gamma}$ and $\Psi_{\neg\Gamma}$ are to be understood as $\Phi \wedge \neg\bigwedge\Gamma$ and $\Psi \wedge \neg\bigwedge\Gamma$. This adornment, which is justified by Lemma 2, is repeated as long as the until formula is in focus. Whenever $\forall$ changes mind, an adorned until subformula $\Phi_{\neg\Gamma_1 \wedge \ldots \wedge \neg\Gamma_n} U \Psi_{\neg\Gamma_1 \wedge \ldots \wedge \neg\Gamma_n}$ loses its adornment and is returned to its intended interpretation $\Phi U \Psi$. Now it is easy to see that $\forall$ can never win. Condition 1 of the winning condition for $\forall$ can not be reached because $\exists$ preserves satisfiability. And condition 2, the repeat position, cannot occur because $\models \Phi_{\neg\Gamma_1 \wedge \ldots \wedge \neg\Gamma_n} U \Psi_{\neg\Gamma_1 \wedge \ldots \wedge \neg\Gamma_n} \rightarrow \neg\bigwedge\Gamma_i$. □

Proposition 3 *The complexity of deciding the winner of $\mathsf{G}(\Phi_0)$ is in PSPACE.*

Proof: Consider the tree of all plays in $\mathsf{G}(\Phi_0)$ where the position of the focus is completely determined by the strategy described in the proof of Proposition 1, above. Player $\exists$ wins $\mathsf{G}(\Phi_0)$ iff there exists a path in this tree such that $\exists$ wins the play of this path. An algorithm P can nondeterministically choose this path. The required space is polynomial in the size of the input. P only has to store a counter and two configurations: the actual one which gets overwritten every time a new game rule is applied, and the one which is repeated in case $\exists$ wins the play with her winning condition 2 or 3. The latter can be chosen nondeterministically, too, and gets deleted every time the rule change is applied. The counter is needed to terminate the algorithm if it did not find a repeat after $|\mathrm{Sub}(\Phi_0)| * 2^{|\mathrm{Sub}(\Phi_0)|}$ configurations. Notice that the size of the counter also is polynomial in the length of the input $|\Phi_0|$. Hence by Savitch's Theorem the problem can be solved in PSPACE. □

4 A complete axiomatisation for LTL

The game theoretic characterisation of satisfiability offers a simple basis for extracting a complete axiom system for LTL. Given an axiom system A a formula Φ is A-consistent if A $\nvdash \neg\Phi$. The axiom system A is complete provided that for any Φ if Φ is A-consistent then Φ has a model. In this framework this becomes

(*) if Φ is A-consistent then $\exists$ wins the game $\mathsf{G}(\Phi)$.

The axiom system A for LTL is presented in Figure 2. The axioms and rules were developed with the proof of (*) in mind. Axioms 1-6 and the rules MP and XGen provide "local" justifications for the rules of the focus game for LTL, and axiom 7 and the rule Rel capture $\exists$'s winning strategy.

Theorem 1 *The axiom system* A *is sound and complete for LTL.*

Axioms

1. any tautology instance
2. $\Phi U \Psi \rightarrow \Psi \vee (\Phi \wedge X(\Phi U \Psi))$
3. $\Phi R \Psi \rightarrow \Psi \wedge (\Phi \vee X(\Phi R \Psi))$
4. $X\neg\Phi \leftrightarrow \neg X\Phi$
5. $X\Phi \wedge X\Psi \rightarrow X(\Phi \wedge \Psi)$
6. $X(\Phi \rightarrow \Psi) \rightarrow X\Phi \rightarrow X\Psi$
7. $\neg(\Phi R \Psi) \leftrightarrow \neg\Phi U \neg\Psi$

Rules

MP if $\vdash \Phi$ and $\vdash \Phi \rightarrow \Psi$ then $\vdash \Psi$

XGen if $\vdash \Phi$ then $\vdash X\Phi$

Rel if $\vdash \Phi' \rightarrow (\Psi \wedge (\Phi \vee X((\Phi \vee \Phi')R(\Psi \vee \Phi'))))$ then $\vdash \Phi' \rightarrow (\Phi R \Psi)$

Figure 2. The axiom system A

Proof: Soundness of A is straightforward. Each axiom is valid and each rule preserves validity. The interesting case is the rule Rel, whose soundness was proved in lemma 2 of the previous section. For completeness of A we establish $(*)$, if Φ_0 is A-consistent then $\exists$ wins the game $G(\Phi_0)$. The proof is similar to Proposition 2 of the previous section. Given a finite A-consistent set of LTL formulas we show that any player $\forall$ move or other move in Figure 1 preserves A-consistency, and that player $\exists$ can preserve A-consistency when she moves. If $\Gamma, \Phi_1 \vee \Phi_2$ is A-consistent then Γ, Φ_i is A-consistent for some i by axiom 1, and the rule MP. Axioms 2 and 3 are needed for the fixed point unfolding moves. Axioms 4-6 and rule XGen are required for the next move. If $\Phi_1, \ldots, \Phi_m$ is not A-consistent then $\mathsf{A} \vdash \Phi_1 \wedge \ldots \wedge \Phi_{m-1} \rightarrow \neg\Phi_m$ and so $\mathsf{A} \vdash X\Phi_1 \wedge \ldots \wedge X\Phi_{m-1} \rightarrow \neg X\Phi_m$ using XGen and axioms 6, 5 and one half of 4. Finally rule Rel is used to capture $\exists$'s winning strategy. If the position is $[\Phi U \Psi], \Gamma$ and $\Gamma, \Phi U \Psi$ is A-consistent then by rule Rel, the other half of axiom 4 and axiom 7 $\Gamma, \Psi \vee (\Phi \wedge X(\Phi_{\neg\Gamma} U \Psi_{\neg\Gamma}))$ is A-consistent. □

In [7] soundness and completeness of the following axiom system DUX for LTL is proved using maximal consistent sets of formulas[2].

[2]A4, A5 and U2 as presented here differ slightly from their original form which is due to the different semantics of the G and U operator used there.

A1. $\mathtt{ff}R(\Phi \rightarrow \Psi) \rightarrow (\mathtt{ff}R\Phi \rightarrow \mathtt{ff}R\Psi)$

A2. $X(\neg\Phi) \leftrightarrow \neg X\Phi$

A3. $X(\Phi \rightarrow \Psi) \rightarrow (X\Phi \rightarrow X\Psi)$

A4. $\mathtt{ff}R\Phi \rightarrow \Phi \wedge X(\mathtt{ff}R\Phi)$

A5. $\mathtt{ff}R(\Phi \wedge X\Phi) \rightarrow (\Phi \rightarrow \mathtt{ff}R\Phi)$

U1. $\Phi U \Psi \rightarrow F\Psi$

U2. $\Phi U \Psi \leftrightarrow \Psi \vee (\Phi \wedge X(\Phi U \Psi))$

R1. any tautology instance

R2. if $\vdash \Phi$ and $\vdash \Phi \rightarrow \Psi$ then $\vdash \Psi$

R3. if $\vdash \Psi$ then $\vdash \mathtt{ff}R\Psi$

Soundness of DUX and completeness of A ensure that, if DUX $\vdash \Phi$ then A $\vdash \Phi$. However, it is also interesting to compare the two axiomatisations in details.

Axioms and rules A2, A3, U2, R1 and R2 are present in A. A4 is an instance of axioms 3 and U1 simply reflects an abbreviation. R3 can be simulated in A as follows. Suppose there is a proof using R3. Then there is a shorter proof of Ψ in DUX for which by hypothesis there is an A-proof, too. Instantiate Rel with $\Phi' = \mathtt{tt}$ and $\Phi = \mathtt{ff}$. This proves $\vdash \mathtt{ff}R\Psi$ if $\vdash \Psi \wedge X\mathtt{tt}$ is provable. But this can be done using the hypothesis, axiom 1 and rule XGen.

The remaining axioms A1 and A5 are more complicated to prove in A. A simple way is to show that $\forall$ wins the focus game on the negations of these axioms. The game rules and winning conditions resemble the axioms and rules of A which are needed for the proof. We show this for A5. The negation of this axiom is $\Phi \wedge (\mathtt{ff}R(\Phi \wedge X\Phi)) \wedge (\mathtt{tt}U\neg\Phi)$. Let $\Phi' = \Phi \wedge (\mathtt{ff}R(\Phi \wedge X\Phi))$.

$$\dfrac{\dfrac{\dfrac{\Phi, \mathtt{ff}R(\Phi \wedge X\Phi), [\mathtt{tt}U\neg\Phi]}{\Phi, X\Phi, X(\mathtt{ff}R(\Phi \wedge X\Phi)), [\neg\Phi \vee X(\mathtt{tt}_{\neg\Phi'}U\neg\Phi_{\neg\Phi'}]}}{\Phi, X\Phi, X(\mathtt{ff}R(\Phi \wedge X\Phi)), [X(\mathtt{tt}_{\neg\Phi'}U\neg\Phi_{\neg\Phi'}]}}{\Phi, \mathtt{ff}R(\Phi \wedge X\Phi), [\mathtt{tt}_{\neg\Phi'}U\neg\Phi_{\neg\Phi'}]}$$

The game rules used are the unfolding of R, the adorned unfolding of U, the disjunctive choice and the next rule. Player $\forall$ wins with winning condition 2. Therefore the axioms and rules needed to prove A5 are 1 and MP (for $\vee$), 2 and 3 (for the unfoldings), 4 – 6, XGen (for next), 7 (to reason about the negation of A5), and Rel to describe the winning condition.

5 CTL

In this section we define focus games for CTL. Again we present CTL in positive form. Formulas of CTL are built from Prop, the boolean connectives $\vee$ and $\wedge$, the two unary temporal operators QX and the four binary temporal operators $Q(\ldots U\ldots)$, $Q(\ldots R\ldots)$ where $Q \in \{E, A\}$. E is the "some paths" quantifier and A is the "for all paths" quantifier.

A Kripke model for CTL formulas consists of a set of states S, a binary transition relation R which is total (for all $s \in \mathsf{S}$ there is a $t \in \mathsf{S}$ such that $s\mathsf{R}t$) and a valuation which assigns to each state $s \in \mathsf{S}$ a maximal consistent set of atomic formulas in Prop. The semantics defines when a state s satisfies a formula Φ, $s \models \Phi$, and it appeals to full paths from a state s_0 which is an ω-sequence of states $s_0 s_1 \ldots$ such that $s_i \mathsf{R} s_{i+1}$ for each $i \geq 0$. In the case of $q \in$ Prop, $s \models q$ iff q belongs to the valuation of s. The clauses for the boolean connectives are as usual. The remaining clauses are as follows.

$s \models EX\Phi$	iff	$\exists t.\ s\mathsf{R}t$ and $t \models \Phi$
$s \models AX\Phi$	iff	$\forall t.$ if $s\mathsf{R}t$ then $t \models \Phi$
$s_0 \models E(\Phi U \Psi)$	iff	$\exists$ full path $s_0 s_1 \ldots\ \exists i \geq 0.\ s_i \models \Psi$ and $\forall j : 0 \leq j < i.\ s_j \models \Phi$
$s_0 \models A(\Phi U \Psi)$	iff	$\forall$ full paths $s_0 s_1 \ldots\ \exists i \geq 0.\ s_i \models \Psi$ and $\forall j : 0 \leq j < i.\ s_j \models \Phi$
$s_0 \models E(\Phi R \Psi)$	iff	$\exists$ full path $s_0 s_1 \ldots\ \forall i \geq 0.\ s_i \models \Psi$ or $\exists j : 0 \leq j < i.\ s_j \models \Phi$
$s_0 \models A(\Phi U \Psi)$	iff	$\forall$ full paths $s_0 s_1 \ldots\ \forall i \geq 0.\ s_i \models \Psi$ or $\exists j : 0 \leq j < i.\ s_j \models \Phi$

The semantics of until and release formulas are determined by their fixed point definitions. $Q(\Phi U \Psi)$ is the least solution to $\alpha = \Psi \vee (\Phi \wedge QX\alpha)$ and $Q(\Phi R \Psi)$ is the largest solution to $\alpha = \Psi \wedge (\Phi \vee QX\alpha)$.

We now define the focus game $\mathsf{G}'(\Phi_0)$ for a CTL formula Φ_0. As with the LTL game, a position in a play of $\mathsf{G}'(\Phi_0)$ is an element $[\Phi], \Gamma$ where $\Phi \in \mathrm{Sub}(\Phi_0)$ and $\Gamma \subseteq \mathrm{Sub}(\Phi_0) - \{\Phi\}$, and a play is a sequence of positions $P_0 P_1 \ldots P_n$ where P_0 is the initial position $[\Phi_0]$. The change in position P_i to P_{i+1} is determined by one of the moves of Figure 3. Again they are divided into three groups. First are rules for $\exists$ who chooses disjuncts in and out of focus. Second are the moves for player $\forall$ who chooses which conjunct remains in focus and who also can change focus with the rule change. Player $\forall$ also chooses the next state when an AX formula is in focus, by choosing a single $EX\Psi_j$, if there is one: we include here the case where $l = 0$ and $\forall$ does not have any choice. Finally, there are the remaining moves which do not involve any choices, and so neither player is responsible for them. These include the fixed point unfolding of until and release in and out of focus, the removal of $\wedge$ out of focus and the next state rule

Player $\exists$

$$\frac{[\Phi_0 \vee \Phi_1], \Gamma}{[\Phi_i], \Gamma} \qquad \frac{[\Phi], \Phi_0 \vee \Phi_1, \Gamma}{[\Phi], \Phi_i, \Gamma}$$

Player $\forall$

$$\frac{[\Phi_0 \wedge \Phi_1], \Gamma}{[\Phi_i], \Phi_{1-i}, \Gamma} \qquad \frac{[\Phi], \Psi, \Gamma}{[\Psi], \Phi, \Gamma}\ \text{change}$$

$$\frac{[AX\Phi_1], \ldots, AX\Phi_n, EX\Psi_1, \ldots EX\Psi_l, q_1, \ldots, q_m}{[\Phi_1], \ldots, \Phi_n, \Psi_j}\ \text{next}$$

Other moves

$$\frac{[Q(\Phi U \Psi)], \Gamma}{[\Psi \vee (\Phi \wedge QXQ(\Phi U \Psi))], \Gamma}$$

$$\frac{[\Phi'], Q(\Phi U \Psi), \Gamma}{[\Phi'], \Psi \vee (\Phi \wedge QXQ(\Phi U \Psi)), \Gamma}$$

$$\frac{[Q(\Phi R \Psi)], \Gamma}{[\Psi \wedge (\Phi \vee QXQ(\Phi R \Psi))], \Gamma}$$

$$\frac{[\Phi'], Q(\Phi R \Psi), \Gamma}{[\Phi'], \Psi \wedge (\Phi \vee QXQ(\Phi R \Psi)), \Gamma}$$

$$\frac{[\Phi], \Phi_0 \wedge \Phi_1, \Gamma}{[\Phi], \Phi_0, \Phi_1, \Gamma}$$

$$\frac{[EX\Psi_1], \ldots, EX\Psi_l, AX\Phi_1, \ldots, AX\Phi_n, q_1, \ldots, q_m}{[\Psi_1], \Phi_1 \ldots, \Phi_n}\ \text{next}$$

Figure 3. CTL Game moves

when an EX formula is in focus. The winning conditions for a player are almost identical to the LTL game.

Definition 1 Player $\forall$ wins the play $P_0, \ldots, P_n$ if

1. P_n is $[q], \Gamma$ and (q is $\mathtt{ff}$ or $\neg q \in \Gamma$) or
2. P_n is $[Q(\Phi U \Psi)], \Gamma$ and for some $i < n$ the position P_i is $[Q(\Phi U \Psi)], \Gamma$ and between $P_i \ldots P_n$ player $\forall$ has not applied the rule change.

Definition 2 Player $\exists$ wins the play $P_0, \ldots, P_n$ if

1. P_n is $[q_1], \ldots, q_n$ and $\{q_1, \ldots, q_n\}$ is satisfiable or
2. P_n is $[Q(\Phi R \Psi)], \Gamma$ and for some $i < n$ the position P_i is $[Q(\Phi R \Psi)], \Gamma$ or
3. P_n is $[\Phi], \Gamma$ and for some $i < n$ the position P_i is $[\Phi], \Gamma$ and between $P_i \ldots P_n$ player $\forall$ has applied the rule change.

Facts 1 and 2 of Section 3 also hold for CTL games. A main result is again the game characterisation of satisfiability.

Proposition 1 *$\exists$ wins the game $\mathsf{G}'(\Phi_0)$ iff Φ_0 is satisfiable.*

Proof: Assume $\exists$ wins the game $\mathsf{G}'(\Phi_0)$. The proof is similar to that of Proposition 1 of Section 3, except that all "next" state choices are examined, and so we have a tree of plays instead of a single play. Let $Q_1(\Phi_1' U \Psi_1'), \ldots, Q_n(\Phi_n' U \Psi_n')$ be an initial priority list of all until subformulas of Φ_0 in order of decreasing size. Each play in the tree of plays has its own associated current priority list. Player $\forall$ starts with the focus on Φ_0. Once the focus is on an until formula, $Q_i(\Phi_i' U \Psi_i')$, player $\forall$ keeps the focus on it until it is fulfilled (player $\exists$ chooses Ψ_i') or there is branching. At an application of next a play splits into all choices, each with its own priority list. If the focus is on a formula $AX\Phi_1$ then it will be on Φ_1 in all these plays and they each have the same priority list. If the position is $[EX\Psi_1], \ldots, EX\Psi_l, AX\Phi_1, \ldots, AX\Phi_n, q_1, \ldots, q_m$ and l is the current priority list then the focus remains on Ψ_1 in the play with this subformula with list l. Otherwise for each $i > 1$ there is the play where $\forall$ changes focus for the position $\Psi_i, \Phi_1, \ldots, \Phi_n$. If Ψ_1 is $E(\Phi_j' U \Psi_j')$ then this formula is moved to the end of the priority list l_i and $\forall$ chooses as focus the earliest until formula in l_i present in the position $EX\Psi_i, AX\Phi_1, \ldots, AX\Phi_n$, if this is possible. This argument is repeated. By assumption player $\exists$ wins the finite tree of plays. It is now straightforward to read off a Kripke model from this tree of plays where Φ_0 is true at the initial state.

For the converse assume that Φ_0 is satisfiable. We show that $\exists$ has a winning strategy for the game $\mathsf{G}'(\Phi_0)$. We use the fact that for each $Q \in \{A, E\}$ if $\Phi' \wedge Q(\Phi U \Psi)$ is satisfiable then $\Phi' \wedge (\Psi \vee (\Phi \wedge QXQ(\Phi \wedge \neg\Phi' U \Psi \wedge \neg\Phi')))$ is satisfiable. So the interpretation of $Q(\Phi U \Psi)$ can be adorned whenever it is unfolded in focus as with Proposition 2 of Section 3. □

One important difference with LTL is the complexity of checking the winner of a game $\mathsf{G}'(\Phi_0)$, because of branching choices for $\forall$.

Proposition 2 *The complexity of deciding the winner of $\mathsf{G}'(\Phi_0)$ is in EXPTIME.*

Proof: The proof is very similar to that of Proposition 3 of Section 3. However, the tree of all plays is now an and-or tree because of player $\forall$'s choices using rule next. Therefore the polynomial space algorithm deciding the winner of $\mathsf{G}'(\Phi_0)$ is alternating instead of nondeterministic. By [3] the problem is therefore in EXPTIME. □

6 A complete axiomatisation for CTL

The game theoretic characterisation of CTL satisfiability also allows one to extract a sound and complete axiom system for CTL, the system B in Figure 4.

Theorem 1 *The axiom system B is sound and complete for CTL.*

Proof: Soundness of B is straightforward. The most interesting cases are soundness of ARel and ERel rules, and in the case of ERel the rule captures "limit closure". For completeness of B, the proof is similar to Theorem 1 of Section 4. If Φ_0 is B-consistent then player $\exists$ wins the game $\mathsf{G}'(\Phi_0)$. Given a finite B-consistent set of formulas, any move by player $\forall$ or other move in Figure 1 preserves B-consistency. The important cases are the next state rules. Assume $\Phi_1, \ldots, \Phi_n, \Psi_j$ is not B-consistent, and so $\mathsf{B} \vdash \Phi_1 \wedge \ldots \wedge \Phi_n \rightarrow \neg\Psi_j$. So by AXGen and axioms 9,8 and 6 $\mathsf{B} \vdash AX\Phi_1 \wedge \ldots \wedge AX\Phi_n \rightarrow \neg EX\Psi_j$ (and using 7 instead of 6 one deals with the case when $l = 0$). Finally the ARel and ERel rules are used to capture $\exists$'s winning strategy. □

In [5] soundness and completeness of the following axiom system for CTL is proved using tableaux.

Ax1. any tautology instance

Ax2. $EF\Phi \leftrightarrow E(\mathtt{tt} U \Phi)$

Ax3. $AF\Phi \leftrightarrow A(\mathtt{tt} U \Phi)$

Ax4. $EX(\Phi \vee \Psi) \leftrightarrow EX\Phi \vee EX\Psi$

Ax5. $AX\Phi \leftrightarrow \neg EX\neg\Phi$

Ax6. $E(\Phi U \Psi) \leftrightarrow \Psi \vee (\Phi \wedge EXE(\Phi U \Psi))$

Axioms

1. any tautology instance
2. $E(\Phi U \Psi) \rightarrow \Psi \vee (\Phi \wedge EXE(\Phi U \Psi))$
3. $A(\Phi U \Psi) \rightarrow \Psi \vee (\Phi \wedge AXA(\Phi U \Psi))$
4. $E(\Phi R \Psi) \rightarrow \Psi \wedge (\Phi \vee EXE(\Phi R \Psi))$
5. $A(\Phi R \Psi) \rightarrow \Psi \wedge (\Phi \vee AXA(\Phi R \Psi))$
6. $AX\neg\Phi \leftrightarrow \neg EX\Phi$
7. $AX\neg\Phi \rightarrow \neg AX\Phi$
8. $AX\Phi \wedge AX\Psi \rightarrow AX(\Phi \wedge \Psi)$
9. $AX(\Phi \rightarrow \Psi) \rightarrow AX\Phi \rightarrow AX\Psi$
10. $\neg A(\Phi R \Psi) \leftrightarrow E(\neg\Phi U \neg\Psi)$
11. $\neg E(\Phi R \Psi) \leftrightarrow A(\neg\Phi U \neg\Psi)$

Rules

MP if $\vdash \Phi$ and $\vdash \Phi \rightarrow \Psi$ then $\vdash \Psi$

AXGen if $\vdash \Phi$ then $\vdash AX\Phi$

ERel if $\vdash \Phi' \rightarrow (\Psi \wedge (\Phi \vee EXE((\Phi \vee \Phi')R(\Psi \vee \Phi'))))$ then $\vdash \Phi' \rightarrow E(\Phi R \Psi)$

ARel if $\vdash \Phi' \rightarrow (\Psi \wedge (\Phi \vee AXA((\Phi \vee \Phi')R(\Psi \vee \Phi'))))$ then $\vdash \Phi' \rightarrow A(\Phi R \Psi)$

Figure 4. The axiom system B

Ax7. $A(\Phi U \Psi) \leftrightarrow \Psi \vee (\Phi \wedge AXA(\Phi U \Psi))$

Ax8. $EX\mathtt{tt} \wedge AX\mathtt{tt}$

R1. if $\vdash \Phi \rightarrow \Psi$ then $\vdash EX\Phi \rightarrow EX\Psi$

R2. if $\vdash \Phi' \rightarrow \Psi \wedge EX\Phi'$ then $\vdash \Phi' \rightarrow E(\Phi R \Psi)$

R3. if $\vdash \Phi' \rightarrow \Psi \wedge AX(\Phi' \vee A(\Phi R \Psi))$ then $\vdash \Phi' \rightarrow A(\Phi R \Psi)$

R4. if $\vdash \Phi$ and $\vdash \Phi \rightarrow \Psi$ then $\vdash \Psi$

The same arguments for comparing the two LTL axiomatisations also hold for the two axiomatisations of CTL. Ax1, Ax5 – Ax7, and R4 are already present in B. Ax2 and Ax3 are covered by the abbreviation of F. Ax4 can be proved by a combination of 6 – 9, 1 and MP. 1, AXGen, 7, MP and 6 establish Ax8. Rule R1 is simulated using AXGen, 9, MP, 7 and the hypothesis of having a shorter proof of $\Phi \rightarrow \Psi$ in B. R2 is simulated in the following way. Suppose there is a B-proof of $\Phi' \rightarrow \Psi \wedge EX\Phi'$. Then, by 4, 1, and MP there is also a proof of $\Phi' \rightarrow \Psi \wedge (\Phi \vee EXE((\Phi \vee \Phi')R(\Psi \vee \Phi')))$ for any Φ. Using ERel yields a proof of $\Phi' \rightarrow E(\Phi R \Psi)$. Simulating R3 is similar.

7 Conclusion

We have introduced a game theoretic approach to satisfiability checking of LTL and CTL. It remains to be seen if focus games extend to richer logics such as CTL* and modal μ-calculus. In [12] it was shown that focus games can also be used to solve the model checking problem for CTL*. The game trees arising there are very similar to the tableau structures used in [2, 1]. However, in order to tackle the problem of deciding whether fixed point constructs are regenerated or reproduced these authors pursue a different strategy. Take the unfolding of $\Phi U \Psi$ for example. While the focus highlights the case that player $\exists$ always chooses the term in which $\Phi U \Psi$ occurs again, a path in the tableaux of [2] is successful if Ψ never occurs after $\Phi U \Psi$. The difference seems to be a point of view only. In the focus games it is checked whether a fixed point construct is regenerated, therefore it is never fulfilled. In the tableau approach it is checked whether it is never fulfilled, therefore it is regenerated.

In [1] the authors define Tableau Büchi Automata which are essentially the same as the tableaux of [2]. As with the focus games, this enables the authors to handle the regeneration problem of fixed points implicitly. Instead of explicitly requiring tableaux to be processed with a depth-first-search, the solution to the regeneration problem is encoded in an acceptance condition, which is in that case a generalised Büchi condition. However, this small difference is the key to the strengthening lemma (Lemma 1 of Section 3)

which underpins the proofs of completeness of the axiomatisations.

A more recent automata theoretic approach to satisfiability and model checking employs alternating automata [16, 11]. Although these appear to be very game theoretic, they rely upon automata over trees which capture the "and" branching, both in the case of the boolean "and" and in the case for CTL of branching through next states. In both cases of LTL and CTL formulas are states of the automata, and transitions are determined by maximal consistent sets of atomic propositions. The acceptance conditions decide acceptable fixed point regeneration. It is not clear if this approach can underpin sound and complete axiomatisations.

References

[1] Bhat, G., and Cleaveland, R. (1996). Efficient model checking via the equational μ-calculus. *Proc. 11th Annual IEEE Symp. on Logic in Computer Science, LICS'96*, 304-312.

[2] Bhat, G., Cleaveland, R., and Grumberg, O. (1995). Efficient on-the-fly model checking for CTL*. *Proc. 10th Annual IEEE Symp. on Logic in Computer Science, LICS'95*, 388-397.

[3] Chandra, A., Kozen, D. and Stockmeyer, L. (1981). Alternation. *Journal of ACM*, **28**, 114-133.

[4] Clark, E., Emerson, E., and Sistla, P. (1986). Automatic verification of finite state concurrent systems using temporal logic. *ACM Trans. on Programming Languages and Systems*, **8**, 244-263.

[5] Emerson, E., and Halpern, J. (1985). Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Comput. System Sci.*, **30**, 1-24.

[6] Emerson, E., and Jutla, C. (1988). The complexity of tree automata and logics of programs. *Procs. 29th IEEE Symp. on Foundations of Comput. Science*, 328-337.

[7] Gabbay, D., Pnueli, A., Shelah, S., and Stavi, J. (1980). The temporal analysis of fairness. *Procs. 7th ACM Symp. on Principles of Programming Languages*, 163-173.

[8] Goldblatt, R. (1992). *Logics of Time and Computation*, CSLI Lecture Notes No. 7.

[9] Hodges, W. (1993). *Model Theory*, Cambridge University Press.

[10] Kozen, D. (1983). Results on propositional mu-calculus. *Theoretical Computer Science*, **27**, 333-354.

[11] Kupferman, O., Vardi, M., and Wolper, P. (2000). An automata-theoretic approach to branching-time model checking. *Journal of ACM*, **47**, 312-360.

[12] Lange, M., and Stirling, C. (2000). Model checking games for CTL*. *Proc. Int. Conf. on Temporal Logic, ICTL'2000.*

[13] Lichtenstein, O., and Pnueli, A. (2000). Propositional temporal logics: decidability and completeness. *Logic Journal of the IGPL*, **8**, 55-85.

[14] Reynolds, M. (2000). An axiomatisation of full computation tree logic. To appear *Journal of Symbolic Logic.*

[15] Stirling, C. (1992). Modal and temporal logics. In *Handbook of Logic in Computer Science*, Vol 2, Oxford University Press, 477-563.

[16] Vardi, M. (1996). An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, **1043**, 238-266.

[17] Vardi, M., and Wolper P. (1994). Reasoning about infinite computations. *Information and Computation*, **115**, 1-37.

[18] Walukiewicz, I. (2000). Completeness of Kozen's axiomatisation of the propositional mu-calculus. *Information and Computation*, **157**, 142-182.

[19] Winskel, G. (1991). A note on model checking the modal nu-calculus, *Theoretical Computer Science*, **83**, 157-167.

Safety and Liveness in Branching Time

Panagiotis Manolios
Computer Sciences Department
University of Texas, Austin, TX, 78712, USA
pete@cs.utexas.edu

Richard Trefler
AT&T Labs-Research
180 Park Ave, P.O. Box 971, Room A013,
Florham Park, NJ, 07932, USA
trefler@research.att.com

Abstract

We extend the Alpern and Schneider linear time characterization of safety and liveness properties to branching time, where properties are sets of trees. We define two closure operators that give rise to the following four extremal types of properties: universally safe, existentially safe, universally live, and existentially live. The distinction between universal and existential properties captures the difference between the CTL path quantifiers A *(for all paths) and* E *(there is a path). We show that every branching time property is the intersection of an existentially safe property and an existentially live property, a universally safe property and a universally live property, and an existentially safe property and a universally live property. We also examine how our closure operators behave on linear time properties.*

We then focus on sets of finitely branching trees and show that our closure operators agree on linear time safety properties. Furthermore, if a set of trees is given implicitly as a Rabin tree automaton, $\mathcal{B}$, we show that it is possible to compute the Rabin automata corresponding to the closures of the language of $\mathcal{B}$. This allows us to effectively compute $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$ such that the language of $\mathcal{B}$ is the intersection of the languages of $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$. As above, $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$ can be chosen so that their languages are existentially safe and existentially live, universally safe and universally live, or existentially safe and universally live.

1 Introduction

Pnueli and Harel introduced the concept of a *reactive system*, a system whose behavior is characterized by nontermination and on-going interaction with an environment over which the system has little control [14]. Many safety critical systems, such as on-board controllers and network protocols, can be modeled as reactive systems and, therefore, the problem of specifying and verifying the correct behavior of reactive systems has become a very active area of research. Linear time properties of reactive systems have been grouped into three categories by Lamport [19]: safety properties, liveness properties, and properties which are neither. Informally, safety properties assert that nothing bad ever happens while liveness properties assert that something good happens eventually. This distinction plays an important role in the analysis of reactive systems since the proof methods employed to check safety properties differ from those used to check liveness properties. For example, proofs of liveness properties frequently require the construction of well-founded relations while safety properties are usually proven by induction on the transition relation. Furthermore, liveness properties often cannot be handled by the automatic proof techniques available for safety properties, *e.g.*, in some infinite state systems it is possible to automatically determine if a safety property can be violated, whereas the existence of a fair computation cannot be determined automatically [1].

In the linear time framework, where properties and the semantics of programs are sets of infinite strings, the distinction between safety and liveness is well understood. Alpern and Schneider [2] give a topological characterization in which safety properties are closed sets and liveness properties are dense sets. They also show that every linear time property can be given as the conjunction of a liveness property and a safety property. These results are well know and now appear in introductory textbooks on distributed systems. The topological characterization has been extended by various researchers, *e.g.*, Gumm has stated the notions of safety and liveness in the more abstract setting of Boolean algebras [13].

In the branching time framework —which includes process algebra and logics such as CTL [7] (which is used by many model checkers and is of great practical importance), CTL* [10], and the μ-calculus [21, 17, 9]— properties and the semantics of programs are sets of infinite trees. While there has been some work on characterizing safety and liveness for the branching time framework [6, 18], we present the first characterization that distinguishes between the CTL

0-7695-1281-X/01 $10.00 © 2001 IEEE

path quantifiers A and E, an essential distinction. In addition, we allow infinitely branching trees; such trees are closely related to considerations of fairness [5, 12, 4] and are useful for modeling input and programs with statements such as $x :=?$ (*i.e.*, non-deterministically assign a number to variable x). We define two closure operators which satisfy the conditions of Gumm [13]. Interestingly, we show that one of the operators defines a topology and the other does not. The closures give rise to four extremal types of properties: *universally safe*, *universally live*, *existentially safe*, and *existentially live*. Universally safe properties are those that correspond to linear time safety properties over all computations while existentially safe properties are those which guarantee at least one safe computation. In a similar manner, universally live and existentially live properties distinguish between linear time liveness properties over all and over some computations. For example, the CTL properties $\mathsf{AG}P$ —along every computation all states satisfy P— is a universally safe property, while $\mathsf{EG}P$ —there is a computation along which all states satisfy P— is an existentially safe property.

The paper is organized as follows: in the next section the basic notations and some preliminaries are given. Section 3 contains a review of the linear time results as well as the definitions of prefixes of trees, our closure operators, and safety and liveness in branching time. Section 3 also includes the results regarding the decomposition of properties into the extremal properties as well as some examples taken from Rem [22]. In Section 4 we consider finitely branching trees and show that for any linear time safety property h, $\mathsf{A}h$ and $\mathsf{E}h$ are both universally safe and existentially safe. In addition, for any linear time liveness property h, $\mathsf{A}h$ is universally live and $\mathsf{E}h$ is both universally live and existentially live. We further specialize our results to properties expressible as Rabin tree automata and show that if a set of trees is given implicitly as a Rabin tree automaton, $\mathcal{B}$, it is possible to effectively compute $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$ such that the language of $\mathcal{B}$ is the intersection of the languages of $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$, where $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$ can be chosen so that their languages are existentially safe and existentially live, universally safe and universally live, or existentially safe and universally live. Finally, Section 5 contains a brief conclusion and comparison with other work.

2 Preliminaries

$\mathbb{N}$ and ω both denote the natural numbers, *i.e.*, $\{0, 1, \ldots\}$. $[i..j]$ denotes the set $\{k \in \mathbb{N} : i \leq k \leq j\}$; $Dom.f$ denotes the domain of function f. Function application is sometimes denoted by an infix dot "." and is right associative. $\langle Qx : r : b\rangle$ denotes a quantified expression, where Q is the quantifier, x the bound variable, r the range of x (**true** if omitted), and b the body. $\mathcal{P}(S)$ denotes the powerset of S. For a relation R, we write $R|_S$ for R left-restricted to the set S, *i.e.*, $R|_S = \{\langle a, b\rangle : (\langle a, b\rangle \in R) \ \wedge \ (a \in S)\}$. S^* denotes the set of finite sequences over S; S^ω denotes the set of infinite sequences (functions from ω) over S; $S^\infty = S^* \cup S^\omega$. Suppose $s, t \in S^\infty$, $\#s$ denotes the length of s or, equivalently, the cardinality of $Dom.s$; s is a prefix of t ($s \preceq t$) iff $Dom.s \subseteq Dom.t$ and for all $i \in Dom.s$, $s.i = t.i$; s is a proper prefix of t ($s \prec t$) iff $s \preceq t$ and $s \neq t$. A set $U \subseteq S^\infty$ is prefix-closed iff for all $u \in U$ and for all $t \preceq u$, $t \in U$.

From highest to lowest binding power, we have: parentheses, function application, binary relations (*e.g.*, sBw), equality ($=$) and membership ($\in$), conjunction ($\wedge$) and disjunction ($\vee$), implication ($\Rightarrow$), and finally, binary equivalence ($\equiv$). Spacing is used to reinforce binding: more space indicates lower binding.

Throughout this paper Σ denotes a fixed *alphabet*, a nonempty set of symbols. An unlabeled tree is a prefix-closed subset of $\mathbb{N}^*$. A tree w is a pair $\langle \mathsf{W}, \mathrm{w}\rangle$ where W is an unlabeled tree and $\mathrm{w} : \mathsf{W} \to \Sigma$. A tree $\langle \mathsf{W}, \mathrm{w}\rangle$ is *total* if $\mathsf{W} \neq \emptyset$ and for all $\sigma \in \mathsf{W}$, there exists $\rho \in \mathsf{W}$ such that $\sigma \prec \rho$. A tree $\langle \mathsf{W}, \mathrm{w}\rangle$ is *finite-depth* if there exists $n \in \mathbb{N}$ such that for all $\sigma \in \mathsf{W}$, $\#\sigma \leq n$. By A^{tot}, A^{nt}, and A^f we denote the set of total, non-total, and finite-depth trees, respectively. The set of trees is denoted by A^{all}; note $A^{all} = A^{tot} \cup A^{nt}$ and $A^f \subset A^{nt}$. Let $t = \langle \mathsf{W}, \mathrm{w}\rangle$ be a tree. A $p \subseteq \mathsf{W}$ is a path in t iff p is a totally ordered (by $\preceq$), prefix-closed subset of W. Given a tree $\langle \mathsf{W}, \mathrm{w}\rangle$ and a node $\sigma \in \mathsf{W}$ we define the path $\overline{\sigma} = \{\sigma' \in \mathsf{W} : \sigma' \preceq \sigma\}$. We extend w to paths: given path $p = p_0 p_1 \cdots$, $\mathrm{w}(p) = (\mathrm{w}.p_0)(\mathrm{w}.p_1)\cdots$.

We briefly describe CTL, CTL*, and LTL [20] formulae (see [8] for complete details). LTL formula are formed from propositions, boolean connectives and the temporal operators X (next time), F (eventually), G (always) and U (until). LTL formulae define sets of infinite strings of (sets of) propositions. CTL* adds the universal and existential branching operators A and E to the LTL syntax. CTL is formed similarly with the restriction that each LTL temporal operator appear paired with its own path quantifier. CTL* and CTL formulae define sets of infinite depth trees labeled with (sets of) propositions.

3 Safety and Liveness

For the linear time framework, Alpern and Schneider define a closure operator on Σ^ω and show that it defines a topology [2]. Their closure operator, $lcl : \mathcal{P}(\Sigma^\omega) \to \mathcal{P}(\Sigma^\omega)$, is defined as follows: $lcl.T = \{t \in \Sigma^\omega : \langle \forall x : x \prec t : \langle \exists t' \in T :: x \preceq t'\rangle\rangle\}$. Properties are subsets of Σ^ω. Safety properties are defined to be the closed sets induced by lcl and liveness properties are defined to be the dense sets. It is shown that any property P is the intersec-

tion of $lcl.P$, a safety property, and $P \cup \neg(lcl.P)$, a liveness property. Gumm defines safety and liveness in the more abstract setting of Boolean algebras [13]. Given B_1 and B_2, two $\vee$-complete Boolean algebras, and $\varphi : B_1 \rightarrow B_2$, a $\vee$-preserving map, the *closure* $\overline{a}$ of $a \in B_1$ is defined as $\vee\{x \in B_1 : \varphi.a = \varphi.x\}$. For element $e \in B_1$, e is a *safety element* iff $\overline{e} = e$ and e is a *liveness element* iff $\overline{e} = 1$ (1 is the unit (top) element of B_1). It is proved that every element of B_1 is the conjunction of a safety element with a liveness element. We obtain the Alpern Schneider result by setting B_1 to $\langle \mathcal{P}(\Sigma^\omega), \Sigma^\omega, \emptyset, \cup, \cap, \neg \rangle$, B_2 to $\langle \mathcal{P}(\Sigma^*), \Sigma^*, \emptyset, \cup, \cap, \neg \rangle$, and $\varphi.T = \{x \in \Sigma^* : \langle \exists a \in T :: x \preceq a \rangle\}$.

To define safety and liveness properties for branching time, we start by defining what it means to concatenate trees and use this notion to define what it means for one tree to be a prefix of another. We then define two prefix operators corresponding to φ above. The closure, safety elements, and liveness elements are defined as above. We then explore the consequences and show that our characterization captures the intuitive notions of safety and liveness in the branching time framework.

3.1 A Partial Order for Trees

Given trees w and x, we define a preliminary notion of tree concatenation, denoted $w \cdot x$.

Definition 1 *Let* $w = \langle \mathsf{W}, \mathsf{w} \rangle$ *and* $x = \langle \mathsf{X}, \mathsf{x} \rangle$ *be trees.* $w \cdot x = \langle \mathsf{W} \cup \mathsf{X}, \mathsf{w} \cup (\mathsf{x}|_{\mathsf{X} \setminus \mathsf{W}}) \rangle$.

Note that $w \cdot x$ is a tree and that this notion of concatenation amounts to superimposing x on w. Unfortunately, the above notion of concatenation turns out not to be what we need. The problem is that it allows us to extend w at non-leaf nodes. Below, we define what it means to be a leaf and then introduce the notion of concatenation we require, where w concatenated with x is denoted by wx.

Definition 2 *Let* $w = \langle \mathsf{W}, \mathsf{w} \rangle$ *be a tree.* $leaf(z, w) \equiv z \in \mathsf{W} \ \wedge \ \neg\langle \exists y \in \mathsf{W} :: z \prec y \rangle$.

Definition 3 *Let* $w = \langle \mathsf{W}, \mathsf{w} \rangle$ *and* $x = \langle \mathsf{X}, \mathsf{x} \rangle$ *be trees. Let* $\mathsf{X}' = \{y \in \mathsf{X} : y \in \mathsf{W} \ \vee \ \langle \exists z : leaf(z, w) : z \prec y \rangle\}$. *Let* $x' = \langle \mathsf{X}', \mathsf{x}|_{\mathsf{X}'} \rangle$. $wx = w \cdot x'$.

Note that wx is a tree; the proof amounts to showing that x' is a tree. We now define what it means for one tree to be a prefix of another.

Definition 4 $x \sqsubseteq y \ \equiv \ \langle \exists z :: xz = y \rangle$

Notice that when restricted to sequences, $\sqsubseteq$ agrees with the usual notion of prefix.

Lemma 1 $x \sqsubseteq y \ \Rightarrow \ wx \sqsubseteq wy$

Lemma 2 $\sqsubseteq$ *is a partial order.*

Note that, due to space restrictions, some of the proofs are omitted.

Elements of $\mathcal{P}(A^{tot})$ are the branching time properties. Note that $\langle \mathcal{P}(A^{tot}), A^{tot}, \emptyset, \cup, \cap, \neg \rangle$ and $\langle \mathcal{P}(A^{all}), A^{all}, \emptyset, \cup, \cap, \neg \rangle$ are Boolean algebras.

3.2 Prefixes and Closures

We define the non-total and finite-depth prefix operators, *npref* and *fpref*, functions from $\mathcal{P}(A^{tot})$ to $\mathcal{P}(A^{all})$, as follows.

Definition 5 $npref.p = \{x \in A^{nt} : \langle \exists y \in p :: x \sqsubseteq y \rangle\}$

Definition 6 $fpref.p = \{x \in A^{f} : \langle \exists y \in p :: x \sqsubseteq y \rangle\}$

The prefix operators correspond to φ, the $\vee$-preserving map described above. The induced closure functions, from $\mathcal{P}(A^{tot})$ to $\mathcal{P}(A^{tot})$, are:

Definition 7 $ncl.p \ = \ \cup\{q \subseteq A^{tot} : npref.q = npref.p\}$

Definition 8 $fcl.p \ = \ \cup\{q \subseteq A^{tot} : fpref.q = fpref.p\}$

The closure functions have the following properties.

Lemma 3 $ncl.p = \{ \ y \in A^{tot} \ : \ \langle \forall x \in A^{nt} : x \sqsubseteq y : x \in npref.p \rangle \ \}$

Lemma 4 $fcl.p = \{ \ y \in A^{tot} \ : \ \langle \forall x \in A^{f} : x \sqsubseteq y : x \in fpref.p \rangle \ \}$

After expanding the definitions of the prefix operators in the above two lemmas, notice that the characterizations of *ncl* and *fcl* are very similar to the definition of *lcl*.

Lemma 5 $p \subseteq ncl.p$ *and* $p \subseteq fcl.p$

Lemma 6 $ncl.ncl.p = ncl.p$ *and* $fcl.fcl.p = fcl.p$

3.3 Safety

We say that a property is a *safety property* if the property is equal to its closure. Since we have two types of closures, we have two types of safety properties: existentially safe (ES) and universally safe (US). The intuition is that the existentially safe properties guarantee at least one computation along which nothing bad happens. The universally safe properties guarantee that nothing bad happens during any computation. This type of distinction is made with the CTL operators E, which existentially quantifies over paths, and A, which universally quantifies over paths. In the sequel, we implicitly extend functions on sets to functions on formulae, be applying the functions to the sets of trees or strings which the formulae define.

Definition 9 *(Existentially Safe)* $p \in ES \equiv p = ncl.p$

Definition 10 *(Universally Safe)* $p \in US \equiv p = fcl.p$

Lemma 7 $ncl.p \subseteq fcl.p$

Proof The domain of the quantifier in the definition of fcl, A^f, is a subset of A^{nt}, the domain of the quantifier in the definition of ncl. □

Lemma 8 $p \in US \Rightarrow p \in ES$

Proof $p \in US \equiv p = fcl.p$, but $p \subseteq ncl.p$ and $ncl.p \subseteq fcl.p$, so $p = ncl.p$, *i.e.*, $p \in ES$. □

Lemma 9 $ncl.fcl.p = fcl.p$

Proof $fcl.p \subseteq ncl.fcl.p \subseteq fcl.fcl.p = fcl.p$ □

We note that $fcl.ncl.p = ncl.p$ does not hold, for example, when $p = \mathsf{EG}a$ (there exists a path such that every node in the path is labeled by an a), we will see that $ncl.p = p$, but $fcl.p \neq p$.

Lemma 10 $p \subseteq q \Rightarrow ncl.p \subseteq ncl.q \wedge fcl.p \subseteq fcl.q$

Recall that an operator $c : \mathcal{P}(X) \rightarrow \mathcal{P}(X)$ defines a topology on X with closed sets $\{a \subseteq X : c.a = a\}$ iff the following four conditions hold [15]:

- $c.\emptyset = \emptyset$
- $a \subseteq c.a$
- $c.c.a = c.a$
- $c(a \cup b) = c.a \cup c.b$

Therefore, the following lemma shows that fcl defines a topology.

Lemma 11 $fcl.p \cup fcl.q = fcl.(p \cup q)$

Since $ncl.(p \cup q) \subseteq ncl.p \cup ncl.q$ is not a theorem, ncl does not define a topology. This does not cause us any technical difficulties, but it is interesting because lcl, the closure operator in the linear time case, does define a topology. We have the following, however.

Lemma 12 $ncl.p \cup ncl.q \subseteq ncl.(p \cup q)$

3.4 Liveness

We will now define what it means for a property to be a liveness property. A liveness property is one whose closure is the set of all trees. Given our two notions of closure, we have two notions of liveness.

Definition 11 *(Existentially Live)* $p \in EL \equiv ncl.p = A^{tot}$

Definition 12 *(Universally Live)* $p \in UL \equiv fcl.p = A^{tot}$

Lemma 13 $p \in EL \Rightarrow p \in UL$

Proof $p \in EL \equiv ncl.p = A^{tot}$, but since $ncl.p \subseteq fcl.p$, $fcl.p = A^{tot}$, *i.e.*, $p \in UL$ □

Lemma 14 $US \cap UL = \{ A^{tot} \}$

Proof $p \in (US \cap UL) \equiv p = fcl.p \wedge fcl.p = A^{tot} \equiv p = A^{tot}$ □

Lemma 15 $ES \cap EL = \{ A^{tot} \}$

Lemma 16 $US \cap EL = \{ A^{tot} \}$

Proof $p \in (US \cap EL) \equiv p = fcl.p \wedge ncl.p = A^{tot} \equiv p = ncl.p \wedge ncl.p = A^{tot} \equiv p = A^{tot}$ □

Note that $ES \cap UL = \{ A^{tot} \}$ does not hold, *e.g.*, ($\mathsf{AF}a$ means along all futures a eventually holds) let $p = ncl.\mathsf{AF}a$, then $p = ncl.p$ and $fcl.p = A^{tot}$, but $p \neq A^{tot}$.

On account of Lemma 10, we have the following two properties:.

Lemma 17 $p \subseteq q \wedge p \in EL \Rightarrow q \in EL$

Lemma 18 $p \subseteq q \wedge p \in UL \Rightarrow q \in UL$

Lemma 19 $(p \cup \neg ncl.p) \in EL$

Proof $ncl.(p \cup \neg ncl.p) \supseteq ncl.p \cup ncl.(\neg ncl.p) \supseteq ncl.p \cup \neg ncl.p = A^{tot}$ □

Lemma 20 $(p \cup \neg fcl.p) \in UL$

Theorem 1 *Every property is the intersection of: (1) an existentially safe and an existentially live property, (2) a universally safe and a universally live property, and (3) an existentially safe and a universally live property.*

Proof (1). $ncl.p \in ES$ and $(p \cup \neg ncl.p) \in EL$; their intersection yields $ncl.p \cap (p \cup \neg ncl.p) = ncl.p \cap p = p$. (2). $fcl.p \in US$ and $(p \cup \neg fcl.p) \in UL$; their intersection yields $fcl.p \cap (p \cup \neg fcl.p) = fcl.p \cap p = p$. (3). $ncl.p \in ES$ and $(p \cup \neg fcl.p) \in UL$; their intersection yields $ncl.p \cap (p \cup \neg fcl.p) = (ncl.p \cap p) \cup (ncl.p \cap \neg fcl.p) = p \cup (ncl.p \cap \neg fcl.p) = p$, since $ncl.p \subseteq fcl.p$. □

The next theorem shows that certain properties do not correspond to the intersection of a universally safe and an existentially live property.

Theorem 2 *[16] Let Q be a subset of A^{tot} such that $fcl.Q = A^{tot}$ and $ncl.Q \neq A^{tot}$. There do not exists sets $S, L \subseteq A^{tot}$ such that $fcl.S = S$, $ncl.L = A^{tot}$, and $S \cap L = Q$.*

Proof Suppose $S \cap L = Q$, $fcl.S = S$, and $ncl.L = A^{tot}$, then $Q \subseteq S$, which gives $A^{tot} = fcl.Q \subseteq fcl.S$ and hence $S = A^{tot}$. Since $Q = S \cap L$, then $L = Q$ which implies that $ncl.Q = A^{tot}$. □

We will see shortly that the set of trees satisfying the CTL formula $\mathsf{AF}p$ satisfies the preconditions on the previous theorem.

Our decomposition of a property into a safety property and a liveness property is extreme in the following sense.

Lemma 21 *If $(q \in ES \ \vee \ q \in US)$ and $p = (q \cap r)$, then $ncl.p \subseteq q$ and $r \subseteq (p \cup \neg ncl.p)$.*

Proof For the first part note that $p = (q \cap r) \ \Rightarrow \ p \subseteq q \ \Rightarrow \ ncl.p \subseteq ncl.q \ \wedge \ ncl.q \subseteq fcl.q \ \Rightarrow \ ncl.p \subseteq q$, as by assumption $q = ncl.q \ \vee \ q = fcl.q$.

For the other part, we have $(q \cap r) = p$, which by $ncl.p \subseteq q$ (the first part) implies $(ncl.p \cap r) \subseteq p$, which, if we union $\neg ncl.p$ to both sides and simplify the left, implies $(\neg ncl.p \cup r) \subseteq (p \cup \neg ncl.p)$, which implies $r \subseteq (p \cup \neg ncl.p)$ □

Theorem 3 *Let h be an LTL formula which is a safety property, then $fcl.\mathsf{A}h = ncl.\mathsf{A}h = \mathsf{A}h$ and $ncl.\mathsf{E}h = \mathsf{E}h$.*

Proof Suppose $t \in A^{tot}$, $t = \langle T, \tau \rangle$, $t \in fcl.\mathsf{A}h$, and $t \notin \mathsf{A}h$. Then there is some path x in t such that $\tau(x) \notin h$. Since h is a safety property, $lcl.h = h$, this implies that for some $i \in \mathbb{N}$, $\tau(x_0 \cdots x_i)$ cannot be extended to a string in h. Hence for any $u \in A^f$ such that $u \sqsubseteq t$ and includes $x_0 \cdots x_i$, u cannot be extended into a tree v such that $v \in \mathsf{A}h$. Hence there is no such t and $fcl.\mathsf{A}h = \mathsf{A}h$. We also have $\mathsf{A}h \subseteq ncl.\mathsf{A}h \subseteq fcl.\mathsf{A}h = \mathsf{A}h$, so $ncl.\mathsf{A}h = \mathsf{A}h$.

Suppose $t \in A^{tot}$, $t = \langle T, \tau \rangle$, and $t \in ncl.\mathsf{E}h$ and $t \notin \mathsf{E}h$. Since $t \notin \mathsf{E}h$ then for no full path, y, in t is $\tau(y) \in h$. Let x be a path whose prefix $\tau(x_0 \cdots x_i)$ cannot be extended to a string in h. Let $u \in A^{nt}$ be the tree obtained from t by making x_i a leaf (*i.e.*, removing all its descendants). Then u cannot be extended into a tree v such that $v \in \mathsf{E}h$ and hence $t \notin ncl.\mathsf{E}h$. Therefore $ncl.\mathsf{E}h = \mathsf{E}h$. □

The following property shows that *fcl* is not an appropriate closure operator for existentially quantified safety properties of paths. That is, a safety property to which existential quantification is added is not necessarily closed under *fcl*.

Lemma 22 $fcl.\mathsf{EG}P \neq \mathsf{EG}P$

Proof Consider a total tree whose root has an infinite number of children, but every other node has exactly one child. Furthermore, the path through the first child is labeled by $a(\neg a)^\omega$. The path through the second child is labeled by $aa(\neg a)^\omega$ and so on. No path in the tree satisfies $\mathsf{G}a$, so the tree is not in $\mathsf{EG}a$, but any finite depth prefix of the tree can be extended to a tree in $\mathsf{EG}a$. □

Theorem 4 *Let h be an LTL formula which is a liveness property, then $fcl.\mathsf{A}h = A^{tot}$ and $ncl.\mathsf{E}h = fcl.\mathsf{E}h = A^{tot}$.*

Proof h is a liveness property implies that $lcl.h = \{\sigma \in \Sigma^\omega\}$.

Let $t \in A^{tot}$, $t = \langle T, \tau \rangle$, and $u \sqsubseteq t$ such that $u \in A^f$. Consider any full path x in u. $\tau(x)$ is a prefix of some $\sigma \in lcl.h$, hence, x can be extended to a path y such that $\tau(y) \in h$. Therefore $u \in fcl.\mathsf{A}h$. Hence $fcl.\mathsf{A}h = A^{tot}$.

Let $t \in A^{tot}$, $t = \langle T, \tau \rangle$, and $u \sqsubseteq t$, such that $u \in A^{nt}$. If u contains a path σ such that $\tau(\sigma) \in h$ then $t \in ncl.\mathsf{E}h$. Else, consider any full finite path x in u. As in the first proof above, x can be extended to an infinite path $y \in h$ and hence $t \in ncl.\mathsf{E}h$. Therefore $ncl.\mathsf{E}h = A^{tot}$. We also have $A^{tot} = ncl.\mathsf{E}h \subseteq fcl.\mathsf{E}h$, hence, $fcl.\mathsf{E}h = A^{tot}$. □

The following property shows that *ncl* is not an appropriate closure operator for universally quantified liveness properties of paths. That is, given a liveness property of paths, adding universal quantification and taking the *ncl* closure does not necessarily result in the set of all trees.

Lemma 23 $ncl.\mathsf{AF}P \neq A^{tot}$

3.5 Examples

We now take a moment to consider the ramifications of our approach by comparison with Martin Rem's [22] example properties, listed below. Rem's examples are formulated as predicates on t, an infinite (Σ) sequence.

$p0$: **false** (corresponds to $\emptyset$);
$p1$: the first symbol of t is a;
$p2$: the first symbol of t differs from a;
$p3$: the first symbol of t is a, and t contains a symbol that differs from a;
$p4$: the number of a's in t is finite;
$p5$: the number of a's in t is infinite;
$p6$: **true** (corresponds to Σ^ω).

If we are dealing with sequences, $p0$, $p1$, $p2$, and $p6$ are safety properties. The (linear) closure of $p3$ is $p1$, so $p3$ is not a safety property. The closures of $p4$ and $p5$ are both Σ^ω; so they are not safety properties, but they are liveness properties.

Note that if we restrict t to infinite sequences, then both *fcl* and *ncl* agree with *lcl*. In order to examine the above properties in a branching time framework, we will write them down in LTL [20] and CTL*. Note that in translating the above examples to properties over trees there is some ambiguity. In particular, we have translated $p4$ into both $q4a$ and $q4b$ and in fact neither of these translations captures the notion that there are only a finite number of a's in a tree but rather that there are a finite number of a's on a path (on all paths) in the tree.

$q0$:	**false**	**false** (corresponds to $\emptyset$);		
$q1$:	a	a;		
$q2$:	$\neg a$	$\neg a$;		
$q3a$:	$a \wedge \mathsf{F}\neg a$	$\mathsf{A}(a \wedge \mathsf{F}\neg a)$	$\equiv$	$a \wedge \mathsf{AF}\neg a$;
$q3b$:		$\mathsf{E}(a \wedge \mathsf{F}\neg a)$	$\equiv$	$a \wedge \mathsf{EF}\neg a$;
$q4a$:	$\mathsf{FG}\ \neg a$	$\mathsf{A}(\mathsf{FG}\ \neg a)$;		
$q4b$:		$\mathsf{E}(\mathsf{FG}\ \neg a)$;		
$q5a$:	$\mathsf{GF}\ a$	$\mathsf{A}(\mathsf{GF}\ a)$;		
$q5b$:		$\mathsf{E}(\mathsf{GF}\ a)$;		
$q6$:	**true**	**true** (corresponds to A^{tot}).		

Below we give an informal translation of the above CTL* sentences. $q1$ is true of any tree whose root is labeled with a; similarly for $q2$. $q3a$ is true of the trees whose root is labeled with a and along each path have a node labeled with $\neg a$. $q3b$ is true of the trees whose root is labeled with a and along some path have a node labeled with $\neg a$. $q4a$ is true of the trees where along each path, eventually all nodes are labeled with $\neg a$. $q4b$ is true of the trees where along some path, eventually all nodes are labeled with $\neg a$. $q5a$ is true of the trees where along each path, infinitely many nodes are labeled with a. $q5b$ is true of the trees where along some path, infinitely many nodes are labeled with a.

It is not difficult to show that $q0$, $q1$, $q2$, and $q6$ are universally safe (and hence existentially safe).

$fcl.q3a = q1$, as before, but $ncl.q3a \neq q1$ (consider a tree that has at least two paths such that along one of the paths a always holds; this tree is not in $ncl.q3a$). $ncl.q3a \neq q3a$ (trees can be sequences, so $\{y : y \in \Sigma^\omega\} \subseteq ncl.q3a$). $ncl.q3b = q1$ and $fcl.q3b = q1$.

$fcl.q4a = A^{tot}$, as before, but $ncl.q4a \neq A^{tot}$ (consider a tree that has at least two paths such that along one of the paths a always holds; this tree is not in $ncl.q4a$). $ncl.q4a \neq q4a$ (trees can be sequences, so $\{y : y \in \Sigma^\omega\} \subseteq ncl.q4a$). $ncl.q4b = A^{tot}$, so $fcl.q4b = A^{tot}$.

$fcl.q5a = A^{tot}$, as before, but $ncl.q5a \neq A^{tot}$ (consider a tree that has at least two paths such that along one of the paths $\neg a$ always holds; this tree is not in $ncl.q5a$). $ncl.q5a \neq q5a$ (trees can be sequences, so $\{y : y \in \Sigma^\omega\} \subseteq ncl.q5a$). $ncl.q5b = A^{tot}$, so $fcl.q5b = A^{tot}$.

4 Finite Branching Trees

In the previous sections we studied sets of trees that included infinitely branching trees. However, many systems do not have such trees and it is interesting to see what benefits are obtainable when considering only bounded branching structures.

Let $k \in \mathbb{N}$. A tree $\langle \mathsf{W}, \mathsf{w} \rangle$ is a k-branching tree iff for all $\sigma \in \mathsf{W}$ there exists exactly k unique elements of $\mathbb{N}$, $a_0, \ldots, a_{k-1}$, such that $\sigma a_0, \ldots, \sigma a_{k-1} \in \mathsf{W}$. In what follows we consider sets of trees which are k-branching. By $A^{k,tot}$ and $A^{k,f}$ we denote, respectively, the set of k-branching trees and the set of finite trees whose non-leaf nodes have exactly k successors. We carry over the definitions of *ncl* and *fcl* from the previous sections, restricted now to k-branching trees over finite alphabets. Below we show that *ncl* and *fcl* agree on linear time safety properties (recall that $ncl.p \subseteq fcl.p$).

Theorem 5 *Suppose h is a safety property over Σ^ω then $fcl.\mathsf{E}h = ncl.\mathsf{E}h = \mathsf{E}h$ and $fcl.\mathsf{A}h = ncl.\mathsf{A}h = \mathsf{A}h$.*

Proof We have that $\mathsf{E}h \subseteq fcl.\mathsf{E}h$. So suppose $t = \langle T, \tau \rangle \in fcl.\mathsf{E}h$, this means that for all $u \in A^{k,f}$, $u \sqsubseteq t$ implies there is a $t' \in A^{k,tot}$ such that $u \sqsubseteq t'$ and $t' \in \mathsf{E}h$.

We will show that t contains a path p such that $\tau(p) \models h$. Consider the tree $v = \langle V, \phi \rangle \in A^{all}$ defined as follows: $V = \{\sigma \in T : \langle \exists y \in \Sigma^\omega :: \tau(\overline{\sigma})y \models h \rangle\}$ and $\phi.\sigma = \tau.\sigma$. V has an infinite number of nodes as any prefix of t can be extended to a tree in $\mathsf{E}h$. V is also finitely branching, thus, by König's lemma, has an infinite path. For any such infinite path p, $\tau(p) \models h$ because h is a safety property. Since $v \sqsubseteq t$, p is a path in T, hence, $t \in \mathsf{E}h$.

The rest of the proof is along the lines of the proof of Theorem 3. □

For linear time liveness properties, however, *fcl* and *ncl* do not agree, *e.g.*, $fcl.\mathsf{AF}P = A^{k,tot}$ whereas $ncl.\mathsf{AF}P \neq A^{k,tot}$. We do have the following.

Lemma 24 *Suppose h is a liveness property over Σ^ω then $fcl.\mathsf{E}h = ncl.\mathsf{E}h = A^{k,tot}$ and $fcl.\mathsf{A}h = A^{k,tot}$.*

The close relationship between properties of programs and automata has been well documented [24, 8]. In particular, given a finite state Büchi automaton, $\mathcal{B}$, over infinite strings (recall that Büchi automata recognize regular languages of ω-strings), it is possible to decompose $\mathcal{B}$ into automata $\mathcal{B}_S$ and $\mathcal{B}_L$ such that the set of strings accepted by $\mathcal{B}_S$ is a safety property and the set of strings accepted by $\mathcal{B}_L$ is a liveness property [3]. Furthermore, the set of strings accepted by $\mathcal{B}$ is equal to the intersection of the set of strings accepted by $\mathcal{B}_S$ and $\mathcal{B}_L$. We show that a similar result for Rabin tree automata is possible to achieve (recall that Rabin automata recognize regular languages of ω-trees). That is, we show that any set of trees recognizable by a Rabin tree automaton is decomposable into the intersection of: a universally safe set and a universally live set, an existentially safe set and an existentially live set, and an existentially safe set and a universally live set, all of which are Rabin tree automata definable.

A Rabin tree automaton $\mathcal{B} = (\Sigma, Q, q_0, \delta, \Phi)$ on k-ary infinite trees is defined as follows: Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is the start state, $\delta : Q \times \Sigma \to \mathcal{P}(Q^k)$ is the transition relation, and Φ is the accepting condition.

Let $t = \langle \mathsf{W}, \mathsf{w} \rangle \in A^{k,tot}$. A run of $\mathcal{B}$ on t is a Q labeled tree $r = \langle \mathsf{W}, \rho \rangle \in A_Q^{k,tot}$ such that $\rho.\lambda = q_0$ and for all $\sigma \in \mathsf{W}$ and successors $\sigma a_0, \ldots, \sigma a_{k-1} \in \mathsf{W}$, $(\rho.\sigma a_0, \ldots, \rho.\sigma a_{k-1}) \in \delta(\rho.\sigma, \mathsf{w}.\sigma)$. Run r is accepting iff for all infinite paths p in W, $\rho(p) \models \Phi$. $\mathcal{L}(\mathcal{B}) = \{t \in A^{k,tot} :$ there is an accepting run of $\mathcal{B}$ on $t\}$ is the language of $\mathcal{B}$.

The accepting condition, Φ, is given by specifying pairs of sets $(green_i, red_i) \in (\mathcal{P}(Q))^2$ for $i \in [0..m]$, for some m. Φ holds on a path if for some i, some green state is visited infinitely often and all red states are visited finitely often, *i.e.*, $\Phi = \vee_{i \in [1..m]}[(\vee_{g \in green_i} \mathsf{GF}g) \wedge (\wedge_{r \in red_i} \mathsf{FG}\neg r)]$.

For notational convenience, given a Rabin automaton $\mathcal{B} = (\Sigma, Q, q_0, \delta, \Phi)$ we will refer to $\mathcal{B}(q)$, $q \in Q$, as the automaton given by $(\Sigma, Q, q, \delta, \Phi)$. Given automaton $\mathcal{B} = (\Sigma, Q, q_0, \delta, \Phi)$ such that $\mathcal{L}.\mathcal{B} \neq \emptyset$, note that $\mathcal{L}.\mathcal{B} = \mathcal{L}.(\Sigma, Q', q_0, \delta', \Phi)$ where $Q' = \{q \in Q : \mathcal{L}(\mathcal{B}(q)) \neq \emptyset\}$ and δ' is δ restricted to Q'. We define the finite depth closure, *rfcl*, of an automaton as follows: if $\mathcal{L}.\mathcal{B} = \emptyset$, $rfcl.\mathcal{B} = \mathcal{B}$; otherwise, $rfcl.\mathcal{B} = (\Sigma, Q', q_0, \delta', \Phi')$ where $\Phi' = \vee_{q \in Q'} \mathsf{GF}q$ is a condition that holds along all paths and is generated from $\{(Q', \emptyset)\}$.

Lemma 25 $\mathcal{L}(rfcl.\mathcal{B}) = fcl.\mathcal{L}(\mathcal{B})$.

Proof If $\mathcal{L}.\mathcal{B} = \emptyset$, then $\mathcal{L}(rfcl.\mathcal{B}) = fcl(\mathcal{L}.\mathcal{B}) = \emptyset$, so we assume $\mathcal{L} \neq \emptyset$.

Suppose $t = \langle \mathsf{W}, \mathsf{w} \rangle \in \mathcal{L}(rfcl.\mathcal{B})$ then there is an accepting run $r = \langle \mathsf{W}, \rho \rangle$ of $rfcl.\mathcal{B}$ on t. Consider any $u = \langle \mathsf{U}, \upsilon \rangle \in A^{k,f}$ such that $u \sqsubseteq t$; $r' = \langle \mathsf{U}, \rho|_{\mathsf{U}} \rangle$ is a partial run of $rfcl.\mathcal{B}$ on u. By the construction of $rfcl.\mathcal{B}$, each leaf node $\sigma \in \mathsf{U}$ of r' is labeled by a node $\rho.\sigma \in Q'$. This means, however, that $\mathcal{L}.\mathcal{B}(\rho.\sigma) \neq \emptyset$ and therefore that each leaf node is extendible into some tree which is accepted by the automaton node labeling the leaf. This implies that for some tree $t' \in \mathcal{L}.\mathcal{B}$, $u \sqsubseteq t'$ and hence $t \in fcl(\mathcal{L}.\mathcal{B})$.

Suppose $t \in fcl(\mathcal{L}.\mathcal{B})$, then for all $t^i \sqsubseteq t$, where t^i denotes the subtree of t up to level i, there exists u_i such that $t^i \sqsubseteq u_i$ and $u_i \in \mathcal{L}.\mathcal{B}$, hence, there exists run r_i of $\mathcal{B}$ on u_i. We now define a run, r, of $rfcl.\mathcal{B}$ on t. r^i, the subtree of r up to level i, is defined recursively as follows. For the base case, r^0 labels the root by q_0 and $R_0 = \omega$. For the recursive case, choose r^{i+1} so that for infinitely many $j \in R_i, r_j$ labels t^{i+1} by r^{i+1} and $R_{i+1} = R_i \setminus \{j \in R_i : r_j^{i+1} \neq r^{i+1}\}$. Note that for all i, R_i is an infinite set such that for all $j \in R_i, r_j^i = r^i$. This is true for R_0 as all runs label the root q_0. Assuming it is true for R_i, then by definition, if R_{i+1} contains j, $r_j^{i+1} = r^{i+1}$. Since the number of possible labelings of t^{i+1} is finite, by the pigeon-hole principle, an infinite subset of R_i indexes runs that assign the same labeling to t^i. Since the acceptance condition for $rfcl.\mathcal{B}$ is trivially satisfied, we have show that r is a run of $rfcl.\mathcal{B}$ on t. □

The consequence of this is the following:

Theorem 6 *For any Rabin tree automaton, $\mathcal{B}$, there exist effectively derivable Rabin automata $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$ such that $\mathcal{L}.\mathcal{B} = \mathcal{L}.\mathcal{B}_{safe} \cap \mathcal{L}.\mathcal{B}_{live}$ and $\mathcal{L}.\mathcal{B}_{safe}$ is universally safe while $\mathcal{L}.\mathcal{B}_{live}$ is universally live.*

Proof Recall that non-emptiness of Rabin tree automata is decidable and Rabin automata are effectively closed under complementation and union [24]. Thus, $\mathcal{B}_{safe} = rfcl.\mathcal{B}$ and $\mathcal{B}_{live} = \mathcal{B} \cup (A^{k,tot} \setminus rfcl.\mathcal{B})$ can be effectively derived from $\mathcal{B}$. That $\mathcal{B}_{safe}$ is safe follows from the above lemma and $\mathcal{B}_{live}$ is live because for any property P, $P \cup (A^{k,tot} \setminus rfcl.P)$ is live. □

Similarly, it is possible to define the non-total closure of a Rabin automaton, which gives rise to the following theorem.

Theorem 7 *For any Rabin tree automaton, $\mathcal{B}$, there exist effectively derivable Rabin automata $\mathcal{B}_{safe}$ and $\mathcal{B}_{live}$ such that $\mathcal{L}.\mathcal{B} = \mathcal{L}.\mathcal{B}_{safe} \cap \mathcal{L}.\mathcal{B}_{live}$ and $\mathcal{L}.\mathcal{B}_{safe}$ is existentially safe (existentially safe) while $\mathcal{L}.\mathcal{B}_{live}$ is existentially live (universally live).*

5 Conclusion

We have given a computation-tree based semantic characterization of the intuitive notions of safety and liveness. Our characterizations are given in terms of the closures of

sets of trees in a manner analogous to Gumm's [13] generalization of the work of Alpern and Schneider for linear time [2]. In fact, our results when restricted to sets of strings are identical since in that case $ncl.p = fcl.p = lcl.p$. Our approach and examples draw heavily on Rem's very readable presentation [22] of the Alpern and Schneider results.

Decomposing branching time properties into four extremal classes, *viz.*, universally safe, universally live, existentially safe, and existentially live, has allowed a characterization of safety and liveness properties which respects the branching time temporal operators A and E of CTL. In contrast, the work of Bouajjani *et al.* [6] is restricted to the regular trees[1] and does not distinguish between existentially and universally safe. They consider only a single closure operator and choosing either *fcl* or *ncl* as that operator results in either EGP not being a safety property or AFP not being a liveness property. In particular, it is possible to show that EGP is not definable by the class of *safety recognizers* (a restricted class of Rabin tree automata) —see the appendix for a proof— and therefore is not classified as a safety property as defined by Bouajjani *et al.* even under the restriction of finitely branching regular trees.

Possible directions for future work include defining subclasses of safety and liveness formulae and syntactically characterizing them, as has been done in the linear time framework by Sistla [23]. Another question is whether there are efficient model checking algorithms for branching time safety properties (see Kupferman and Vardi [18]).

Acknowledgments

We thank Nils Klarlund for many interesting discussions regarding this work. We also thank an anonymous referee for suggesting, among other things, the adverbs "universally" and "existentially" to distinguish between the two types of safety and liveness properties.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and T. Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*. To appear.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York, 1991.

[5] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, Oct. 1986.

[1] Regular trees are a strict subset of the set of trees [24].

[6] A. Bouajjani, J. C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *18th ICALP, Automata, Languages and Programming*, volume 510 of *LNCS*. Springer-Verlag, 1991.

[7] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, May 1981.

[8] E. A. Emerson. Temporal and modal logic. In van Leeuwen [25], pages 995–1072.

[9] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings 7th International Colloquium on Automata, Languages, and Programming*, volume 85 of *LNCS*. Springer-Verlag, 1981.

[10] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *JACM*, 33(1):151–178, Jan. 1986.

[11] W. Feijen, editor. *Beauty is Our Business*. Springer-Verlag, 1990.

[12] N. Francez. *Fairness*. Springer-Verlag, Berlin, 1986.

[13] H. P. Gumm. Another glance at the alpern-schneider characterization of safety and liveness in concurrent executions. *Information Processing Letters*, 47(6):291–294, Oct. 1993.

[14] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.

[15] J. L. Kelly. *General Topology*. D. Van Nostrand, 1955.

[16] N. Klarlund. personal communication.

[17] D. Kozen. Results on the propositional Mu-Calculus. *Theoretical Computer Science*, pages 334–354, December 1983.

[18] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 172–183. Springer-Verlag, 1999.

[19] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2:125–143, Mar. 1997.

[20] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, 31 Oct.–2 Nov. 1977. IEEE.

[21] V. R. Pratt. A decidable mu-calculus: Preliminary report. In *22nd Annual Symposium on Foundations of Computer Science*, pages 421–427, Nashville, Tennessee, Oct. 1981. IEEE.

[22] M. Rem. A personal perspective of the alpern-schneider characterization of safety and liveness. In Feijen [11], pages 365–372.

[23] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.

[24] W. Thomas. Automata on infinite objects. In van Leeuwen [25], pages 135–192.

[25] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*. Elsevier, Amsterdam, 1990.

A Appendix

All definitions and terminology in the appendix are taken from Bouajjani *et al.* [6]

Definition 13 *A Kripke tree is a tuple* $K = (Q, \Sigma, q_0, R, \pi)$ *where* Q *is a countable set of states,* q_0 *is the initial state,* $R \subseteq Q \times Q$ *is the transition relation (having no cycles and enforcing finite branching),* Σ *is a finite alphabet and* $\pi : Q \to \Sigma$ *is the labeling function.*

Definition 14 *A safety recognizer is a tuple* $S = (\Sigma, W, w_0, \rho)$ *where* W *is a finite set of states,* w_0 *is the initial state and* $\rho \subseteq W \times \Sigma \times \mathcal{P}(W)$ *is the transition relation.*

Definition 15 *A safety recognizer* $S = (\Sigma, W, w_0, \rho)$ *accepts Kripke tree* $K = (Q, \Sigma, q_0, R, \pi)$ *iff there exists* $\lambda : Q \to W$ *such that* $\lambda(q_0) = w_0$ *and for all* $q \in Q$, *there exists* $\Gamma \subseteq W$ *such that* $(\lambda(q), \pi(q), \Gamma) \in \rho$ *and* $\{\lambda(q') : (q, q') \in R\} \subseteq \Gamma$.

Lemma 26 *There is no safety recognizer* S *such that,* S *accepts* K *iff* K *satisfies* $\mathsf{EG}P$.

Proof Assume, to the contrary, that there is such an $S = (\Sigma, W, w_0, \rho)$.

Let $K = (Q, \Sigma, q_0, R, \pi)$ be defined as follows. $\{q : (q_0, q) \in R\} = \{q_1, q_2\}$. Furthermore, $\pi(q_0) = \pi(q_2) = P$ and $\pi(q_1) = \neg P$. Also, suppose K is a total tree and K has one full path (through q_2) which satisfies $\mathsf{G}P$. Since K satisfies $\mathsf{EG}P$ then by assumption K is accepted by S and there exits $\lambda : Q \to W$ such that $\lambda(q_0) = w_0$ and for all $q \in Q$, there exists $\Gamma \subseteq W$ such that $(\lambda(q), \pi(q), \Gamma) \in \rho$ and $\{\lambda(q') : (q, q') \in R\} \subseteq \Gamma$.

Consider $K' = (Q', \Sigma, q_0, R', \pi')$ defined as follows. Intuitively, K' consists of the root of K, the subtree of K rooted at q_1 plus another copy of the subtree rooted at q_1 in place of the subtree rooted at q_2 which has been completely excised. Formally, let $Q_1 = \{q \in Q : q \text{ is a descendant of } q_1 \text{ in } K\}$. Then $Q' = \{q_0\} \cup Q_1 \cup \{q' : q \in Q_1\}$. $(a, b) \in R'$ iff

- $a = q_0$ and $b = q_1$ or q_1' or
- $a, b \in Q_1$ and $(a, b) \in R$ or
- $q, r \in Q_1$, $(q, r) \in R$, $a = q'$, and $b = r'$.

π' is defined as follows: $\pi'(q_0) = \pi(q_0)$; for $q \in Q_1$, $\pi'(q) = \pi(q)$; for $q \in Q_1, \pi'(q') = \pi(q)$. Clearly, K' does not satisfy $\mathsf{EG}P$.

Consider $\lambda' : Q' \to W$ defined as follows. $\lambda'(q_0) = \lambda(q_0)$. For all $q \in Q_1$, $\lambda'(q) = \lambda(q)$ and $\lambda'(q') = \lambda(q)$. Then $\lambda'(q_0) = \lambda(q_0) = w_0$. Also, $(\lambda(q_0), \pi(q_0), \Gamma) \in \rho$ and $\{\lambda(q_1), \lambda(q_2)\} \subseteq \Gamma$, for some Γ. Hence, $\{\lambda'(q_1), \lambda'(q_1')\} \subseteq \Gamma$ and $(\lambda'(q_0), \pi'(q_0), \Gamma) \in \rho$.

Suppose $q \in Q_1$. $(\lambda(q), \pi(q), \Gamma_q) \in \rho$ for some Γ_q and that $\{\lambda(r) : (q, r) \in R\} \subseteq \Gamma_q$. This implies that $(\lambda'(q), \pi'(q), \Gamma_q) \in \rho$ and that $\{\lambda'(r) : (q, r) \in R'\} \subseteq \Gamma_q$. Furthermore $(\lambda'(q'), \pi'(q'), \Gamma_q) \in \rho$ and that $\{\lambda'(r') : (q, r) \in R'\} \subseteq \Gamma_q\}$. Hence, S accepts K', contradicting the assumption that S accepts only those trees satisfying $\mathsf{EG}P$. □

Short Papers

Invited Talk

Semistructured Data: from Practice to Theory

Serge Abiteboul

Abstract

Semistructured data is data that presents some regularity (it is not an image or plain text) but perhaps not as much as some relational data or some ODMG data (the standard of object databases). Such data is becoming increasingly important and, with XML, should become the standard for publishing data on the Web. With XML, the Web is turning into a worldwide, heterogeneous, distributed database. In this paper, we briefly discuss typing and languages for semistructured data and some new issues arising from the context of data management on the Web.

1 Introduction

The amount of data of all kinds available electronically has increased dramatically in recent years. The data resides in different forms, ranging from unstructured data in file systems to highly structured in relational database systems. Data is accessible through a variety of interfaces including Web browsers, database query languages, application-specific interfaces, or data exchange formats. A lot of information can already be found on the Web, sometimes hidden behind forms (the deep Web) or protected by passwords and fire walls. Some of this data is *raw* data, e.g., images or sound. Some is text (e.g., in HTML) allowing access to information via search engines. A lot of this information has some structure, e.g., documents in HTML or XML (the *eXtensible Markup Language*), the forthcoming semistructured standard of the Web.

Semistructured data was first studied in the context of integration of a large volume of data from heterogeneous sources. Data exchange formats, essentially syntax for semistructured data, naturally arose in a number of fields that felt uncomfortable with the lack of flexibility of traditional database systems, e.g., ASN.1. With the popularity of the Web and the choice of XML, such a model, for replacing HTML, the area gained a lot of momentum. Indeed, I like to think of the Web of tomorrow as a gigantic, distributed semistructured database. This is somewhat the vision followed in the Xyleme Project that we initiated at INRIA [20] which aims at building a dynamic warehouse of XML data found on the Web.

The main goal of the present paper is to discuss essential aspects of semistructured data and consider proposals for foundations for such data. We will see that these borrow a lot from computer science theory: database theory, logic and computer science, automata and language theory, type theory.

The paper is organized as follows. In Section 2, we define semistructured data. In the next two sections, we discuss typing and query languages. The separation between these two sections is somewhat arbitrary since the topics are obviously closely related. Most works on typing and queries for semistructured data have focused on single documents or small collections of documents. In a last section, we discuss new challenges that arise from moving to the scale of the Web.

Although the area is rather young, it is very active and the literature it generates keeps growing. For instance, I found 74 references in the DBLP Anthology [8] for "semistructured" and 54 for "semi-structured" (which is why I am using the spelling "semistructured"). I will provide here only few references. Many more can be found in the book [1]. More references on the theory of semistructured data can be found in Vianu's nice survey [17]. One might also want to look at the tutorial on semistructured data and XML by Suciu at VLDB99 [14]. References for databases can be found in [15, 16, 2]. A good entry point for XML (and everything on it) is W3C, the WWW Consortium [18].

2 Semistructured Data

In this section, we make more precise the notion of semistructured data, how such data arises, and describe its main aspects.

Semistructured data is data that presents some regularity (it is not an image or plain text) but perhaps not as much as some relational data or some ODMG data (the standard of object databases). Clearly, this definition is imprecise. For instance, would a BibTex

0-7695-1281-X/01 $10.00 © 2001 IEEE

file be considered structured or semistructured? Indeed, the same piece of information may be viewed as unstructured at some early processing stage, but later become very structured after some analysis has been performed. The first use of the term semistructured (to my knowledge) is in the OEM model [12]. Essentially the same model was proposed simultaneously in [11]. The most popular example of semistructured data today is XML [18]. We will focus on XML here, and present it next (in simplified form).

An example of an XML document is the text given in Figure 1, left. There is an alternative vision of the same document as a tree, also given in the figure. Ignoring details XML has three main components:

Ordered tree (elements and text nodes): An XML piece of data is a tree where leaves are called *text* nodes (grey discs in the figure) and other nodes, the *element* nodes (white discs) may have an unbounded number of ordered children. Each node has a value (a string) attached to it. The value of an element is called a *label* or a *tag*.

Attributes nodes: Element nodes may also have *attributes* (represented by a square in the figure). Each node may have at most one attribute with a given label. Furthermore, the attributes of a node are viewed as unordered.

Graph: A standard trick allows to move to a graph representation. See Figure 2. Some nodes are given identifiers and references to these identifiers may be used in other places of the document.

Ignoring attributes and text, i.e., focusing on the core syntax of XML (i.e., tags), leads to a particular class of context-free languages, see [6]. Let A be the set of opening tags (e.g., $\langle title \rangle$) and $\overline{A}$ the set of closing tags (e.g., $\langle /title \rangle$). Then a *well-formed* XML document is a string of tags of the form $a...\overline{a}$ for some tags a, that is correctly parenthesized. Thus, a strong connection exists between the XML world and languages known in formal language theory under the name of the set of Dyck primes.

What is exactly XML? Three alternative viewpoints are shown in Figure 1.

1. A word: A piece of XML data is a word in some standard language. This is a giant step: one needs only one parser, one browser, one editor, etc.

2. A tree: The same data may be viewed as a tree, the parse tree of the word.

3. An object: It may also be viewed as an object with an interface, e.g., a method *get_parent*, in a standardized application programming interface, namely DOM for *Document Object Model* (the main interface to program applications with XML data).

XML provides three more viewpoints:

1. A document: Data may be displayed with standard Web browsers. For that, we attach a *stylesheet* (in a format called XSL) to an XML string to provide it with a presentation. The simple (and old) idea of separating the data and its presentation is finally coming to the Web.

2. Type data: a type (in a format called DTD - Document Type Definition) or a schema (XML-schema) can be attached to some XML data. (See Section 3.) Now, this is the Web, so one should not expect everybody to use the same tag (e.g., *address*) for the same concept, or the same type for a given tag.

3. Semantics: Once typed information is provided, one can attach semantics to it and describe that semantics. For instance, the Resource Description Framework (RDF) is a standard for publishing semantic descriptions of Web resources. This is leading to the realm of "semantic Web".

Thus XML is reconciling many worlds. In particular, one can view it as the convergence of databases and (hypertext) documents. Viewing XML as text and presentation is central for document management but will be little considered here. We are more concerned with viewing XML as data, or knowledge. Furthermore, we are primarily interested in considering the knowledge available on the Web as a distributed XML database that can be queried like any centralized database.

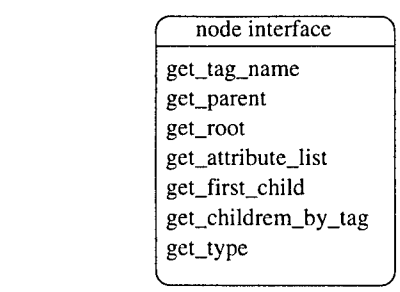
Given this worldwide, distributed, heterogeneous database of semistructured data, a first issue is its management. Can database technology be used? Observe that database systems have been successful because they are easy to use and very efficient. There are many reasons for this: in database system, (i) data is very structured and rigid; (ii) data has a precise known location, generally centralized; (iii) a cost model for queries is available (even if most of the time, it is very rough) to perform optimization; (iv) data can be trusted and is non-contradictory. Every single of these points is defeated for semistructured data on the Web. A second issue is that of formal foundations. Can database theory be used? What else can be used?

3 Typing

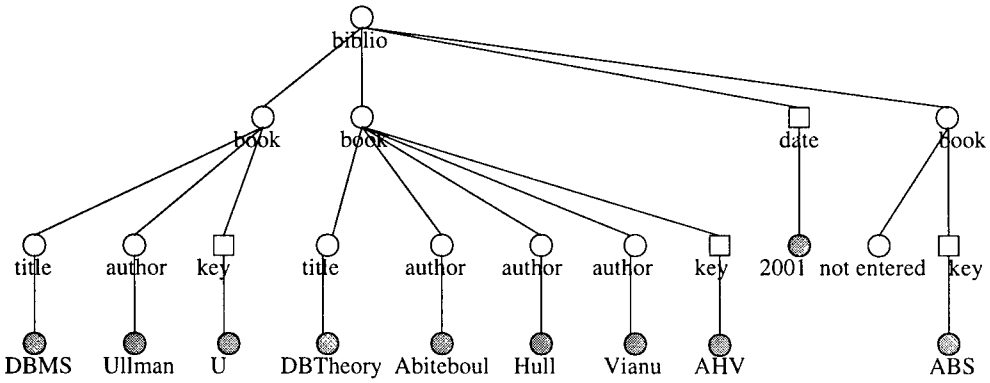
We consider in this section the issue of typing semistructured data.

```
<biblio date="2001">
  <book key="U">
    <title>DBMS</title>
    <author>J.D. Ullman</author>
    </book>
  <book key="AHV">
    <title>DBTheory</title>
    <author>Abiteboul</author>
    <author>Hull</auhor>
    <author>Vianu</author></book>
  <book key="ABS">
    <not entered/>
    </book></biblio>
```

Text View



Object View



Tree View

Figure 1. XML and trees

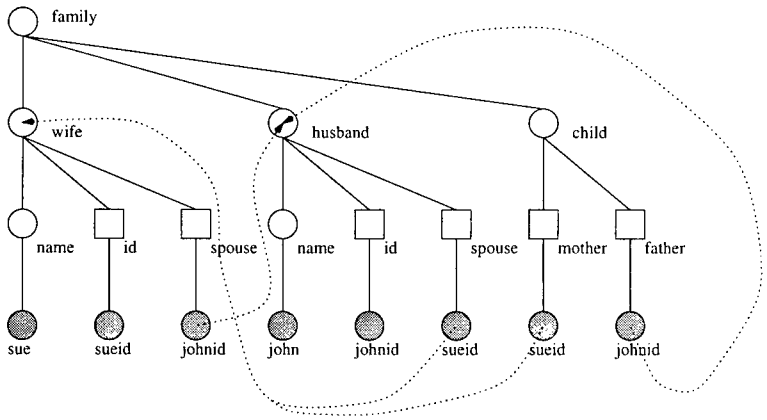


Figure 2. XML and graphs

First, we shall stress that types should not be too constraining in this context. For instance, it is acceptable to have an XML document without type or XML data with portions that are typed and others that are not. More precisely, although the document has some structure, the structure may be irregular (e.g., missing data) and may even violate the type that it is supposed to obey. In traditional databases, data may be large and rapidly evolving whereas types are supposed to be relatively small and stable. This is not true for semistructured data. Indeed, the almost religious distinction between schema and data found in databases is blurred here. Underlying all these aspects is a need for *flexibility*. Flexible typing is not a new notion. For instance, parameterized records have been studied in the context of typed functional languages that allow to type functions applying to records with variable collections of attributes. So, for instance, we may want to see the "type" of a book as:

$$[title, author, editor?, year, \underline{more}]$$

where the question mark means that *editor* is not a compulsory attributes and *more* means that other labels are acceptable there as well .

We next develop some (light) formalism for semistructured data and XML typing. As already mentioned, XML is a syntax. Recall that it is based on opening tags in A and closing tags in $\overline{A}$ with proper parenthesizing. A grammatical approach can be used to define the *type* of a document. More precisely, one can specify that a particular document is *valid* with respect to a certain *Document Type Definition* (DTD). DTDs may be viewed as particular context-free grammars. (An example of a possible DTD for the data in Figure 1 and of the same type in a richer formalism, namely XML-schema are given in Figure 3.) These grammars are special in that each word generated by one such grammar (almost) encodes its parse-tree. More precisely, a DTD specifies for each tag a, a regular expression R_a which tells what can be found between a and $\overline{a}$. An example of DTD (using formal language notation and not XML notation) is as follows:

$$X_a \rightarrow a(X_e^*|X_c)\overline{a} \quad X_e \rightarrow eX_c\overline{e} \quad X_c \rightarrow c\overline{c}$$

where, for instance, the regular expression defining what may be found between a and $\overline{a}$ is $X_e^*|X_c$. A valid word for this DTD is, for instance, $a\ e\ c\ \overline{c}\ \overline{e}\ e\ c\ \overline{c}\ \overline{e}\ \overline{a}$.

An *XML-language* is the set of valid words for a given DTD. It is a context-free language. However, XML-languages enjoy many properties that do not hold in general for context-free languages. For instance, it is not complicated to show that XML-languages are closed under intersection [6]. Note however that the situation is a bit more confusing because the field is still changing rapidly. There are proposals to extend DTDs (e.g., X-schemas) that may modify the kind of results that hold for DTDs as they now stand.

To continue with the issue of semistructured data typing, we may also think of XML as data and adopt a more database-like approach. We may first try to use what is known from the relational database world. It is easy to represent XML data in tables (although this is probably not a good idea for storing it). See Figure 4 for a relational representation of the XML tree of Figure 1. Like in relational dependency theory, first-order logic can be used to express properties on these tables. For *set-oriented* properties, this is a rather convenient formalism. So, for instance, consider the following DTD rule:

$$book \rightarrow \langle book\rangle title(author)^* year\langle/book\rangle$$

Ignoring positions, this can be captured by simple formulas in the style of:

$$\begin{array}{l}\forall b(book(b) \Rightarrow \exists t(title(t) \wedge E(b,t)))\\ \forall b,t,t'(book(b) \wedge title(t) \wedge title(t') \wedge\\ \quad E(b,t) \wedge E(b,t') \Rightarrow t = t')\\ \forall b,x(book(b) \wedge E(b,x) \Rightarrow (title(x) \vee\\ \quad author(x) \vee year(x)))\end{array}$$

As already mentioned, this works fine for set-oriented properties. On the other hand, in a relational representation, the ordering of the children of a node is captured by position and the *list* of these children is not directly available. Furthermore, the tree structure has been encoded/buried into this flat structure. So, many useful properties and queries that typically refer to paths in the tree cannot be directly captured in first-order terms. Following are two examples:

1. regular expressions on the children of a node: DTDs allow to state that, for instance, the sequence of children labels for a node of label a is a word in the language bc^*d. This simple fact is not easy to state in first-order terms.

2. regular expressions for a downward path: given a document d, it is natural to ask for all the elements o such that the labels on the path from the root to o is, e.g., a word in the language $c^*(e|f)c$. Indeed, such features are supported in a language called XPATH that allows to specify complex paths in XML data and is used, in particular, for document presentation. This is also not easy to state in first-order terms.

```
<!DOCTYPE biblio [
<!element biblio (book)*>
<!element book (title, (author)*)|(\#PCDTA)>
<!element title (\#PCDATA)>
<!element author (\#PCDATA)>
<!attlist book key \#PCDATA>
<!attlist biblio date \#PCDATA> ]>
```

Figure 3. Typing XML with DTD

source
&0...

value	
&3	*DBMS*
&5	*Ullman*
...	

E		
&0	&1	1
&1	&2	1
&2	&3	–
&1	&4	2
&4	&5	–...

biblio
&0...

book
&1...

title
&2...

author
&4...

Figure 4. Relational Representation

This naturally suggests the need for recursion and approaches based on fixpoints or proofs (e.g., logic programming and deductive databases).

The two examples we used for illustrating the limitations of first-order logic were based on regular languages. Indeed, approaches based on regular languages and automata techniques seem appropriate in this context and have been investigated. For instance, one can describe paths in the XML tree corresponding to a given DTD with regular languages. This has been used to provide user-friendly graphic interfaces to query such data (in the style of Query-by-Example for relational data). The user navigates through the documents, choosing which *set of nodes* to visit next by selecting a path. It is also natural to describe the type of a document by a tree or a graph. This suggests a definition of typing based on graph homomorphism in the style of graph simulation used, e.g., in program analysis. Last but certainly not least, there have been a series of works on using tree automata to define semistructured data types. Since we will encounter tree automata in the context of queries as well, we postpone their discussion to the next section.

Between all these approaches, there is no clear winner yet and there is still a long way until an analog for semistructured data to dependency theory for relational databases is obtained. The context is much richer and it is likely that foundations for semistructured data typing will be more complex and borrow from several of these approaches. To conclude this discussion on types, we consider two critical use of types in the Web context:

1. Type integration: In a particular application domain, say biology, if each single person publishing his data on the Web uses untyped XML or her own DTD, the construction of a global view of all the information of the Web in the biology domain will have to rely on expensive AI techniques and will probably remain an elusive goal for a long while. On the other hand, if everyone agrees on one DTD (or a small number of DTSs), this integration becomes feasible, see, e.g., [20].

2. Type discovery: As already mentioned, types are often not specified in data found on the Web. However, it is important to be able to understand the structure of data (discover its type) for a number of reasons ranging from query optimization, to explaining the data to users.

4 Logic and Queries

There are many relationships between logic and computer science. One may argue that the most impressive practical application of logic in computer science as of today is relational databases, primarily owing to the algebraization of first-order logic. In a nutshell, this result brings to millions of relational database users an interface to state first-order formulas over a finite structure and get the bindings of variables as answers. Relational database technology has revolutionized access to information. The next revolution may come from query languages for semistructured data, when such data becomes the Web of tomorrow.

Before considering various approaches to query languages for semistructured data, one should note some desired functionalities. First, declarative languages are preferable. The old duality of relational calculus (declarative) vs. relational algebra (operational) survives when we move to semistructured data. However, the distinction is not as clear cut since features like regular expressions (for describing paths) may be viewed both as declarative and procedural/navigational. Then, the language should support information-retrieval-style features such as keyword search. Also, as standard in such context, the language should blur the distinction between schema and data. Finally, since the Web keeps changing, query languages should allow to query these changes. In relational databases, notions such as versions and temporal queries are often supported, see, e.g. [13]. In the Web context, there is growing activity around *query subscriptions* and *continuous queries*. An example of (simple) query subscription is "let me know when a page of this particular site changes". Such services are becoming available on the Web. The underlying technology is related to triggers and active databases [19]. An example of continuous query is "send me, every Wednesday, the list of movies showing in Paris".

We next consider various approaches that have been proposed for querying semistructured data. Everything does not have to be built from scratch. Languages for hierarchical data have been studied for many years. Some of this work has focused on extensions of first-order logic with some *controlled* second-order features, allowing the quantification over sets of values. ("Controlled" here is essential so that query evaluation remains feasible.) From an algebraic/functional viewpoint, this amounts to extending relational algebra (projection, selection, join, etc.) with new operators such as filter, map, comprehension. Logics and algebras have been studied for trees (nested relations) or graphs (complex objects) that can be adapted to semistructured data. For instance, a typical operation, called *nest*, is as follows. Suppose R contains a set of pairs. For each value a, we can group the corresponding values with a *nest* operation. This corresponds to the second-order formula:

$$\{x, Y \mid \exists y(R(x,y)) \wedge \forall y(R(x,y) \Leftrightarrow y \in Y)\}$$

Several query languages (typically using an SQL flavor) have been proposed for semistructured data. For XML alone, there is a flurry of recent competing proposals. Many of them, originating in academia, are arguing in favor of extending OQL [7], a reasonably clean functional language that was adopted as the standard for object databases. Others, mostly from industry, lobby for ad-hoc (one might say inelegant or dirty?) extensions of SQL. At the core of these extensions, one finds tree-pattern matching and tree rewriting. Indeed, one can view these languages as extensions of first-order logic with tree-pattern matching and some form of regular path expressions. Lorel [3] was, I believe, the first OQL extension proposed for semistructured data. An example of query, using a Lorel-like syntax, is:

```
select X/title, X/author
from   X in MyBibliography/biblio/book
where  X/author="Ullman" and X/year="1986"
```

The pattern here is a tree with two branches. A matching pattern consists of a root (the given document *MyBibliography* labeled *biblio*), a child labeled *book* with two children labeled *author* and *year* with appropriate values, "Ullman" and "1986", respectively. Each such pattern that is found produces an element of the answer with a *title* and an *author*. As previously mentioned, regular expressions and keyword search may come into the picture as in, for instance:

```
select X/title, X/author
from   X in MyBibliography/biblio/book
where  X/author="Ullman" and
       X/text//example contains "XML"
```

This asks for the books by Ullman that mention the word XML in an example. In the query, the symbol "/" is used to denote children of a node whereas "//" is used for descendants.

Another line of investigation for query languages is based on structural recursion. For instance, XSLT, a transformation language supported by the Web consortium, allows to specify iterators and tree rewriting patterns to apply on a given document. (It has been claimed recently that XSLT is Turing complete.)

Finally, two related approaches have been recently considered: tree transducers (see, e.g., [10]) and *k-pebble transducers* [9].

Tree transducers The starting point is the view of an XML document as a tree. This suggests using devices over trees and in particular tree transducers. The transducers that are considered are not quite standard in that trees have unbounded fan-out (the official terminology is *unranked*) and a query does not accept/reject the tree but returns a result, typically a set of nodes in the tree. The automaton uses top-down and bottom-up state transitions. A node is selected depending on the state of the automaton when visiting the node and the label of the node. This approach is interesting also because of the equivalence of tree automata and *monadic* second-order logic.

K-pebble transducers These devices subsume most aspects of query languages and typing previously introduced for semistructured data. They are variations of tree automata that we will not define here. Intuitively, a k-pebble transducers performs a computation on a tree. It uses a stack of pebbles to describe the state of the computation so far. The pebbles are installed on tree vertices. At some point of the computation, the transducer may span several parallel computations for the different children of the current node and put them in charge of computing different parts of the result. Figure 5 gives an intermediary state of a computation. Two parallel computations are going on. Each is in charge of computing one subtree of the root of the result.

5 In Place of Conclusion

Essential differences with traditional databases arise from the nature of the Web: (i) its size; (ii) its distributed nature; (iii) the absence of centralized control. This suggests new research directions. To conclude, we mention next (somewhat arbitrarily) five such directions.

Complexity: the complexity of relational queries has been extensively studied. Theory has gone a long way from showing *logspace* and AC^0 bounds for relational algebra to, for instance, obtaining many results for recursive languages (datalog, fixpoint). What is new? A lot when we consider the Web. Logspace at the scale of the Web is simply too much. There is clearly a need for new notions of feasibility in this context.

Computability: Consider a Web crawler. It is essentially an infinite computation. By the time it takes to read the entire Web, a large portion of the data that has been read has already changed, some has disappeared, new data arrived. So, strictly speaking, some queries such as *give me the list of URLs pointing to my homepage at the exact instant* can simply not be answered. Thus, even the notion of computability has to be reconsidered, see [5] and should encompass *infinite* computations.

A world of changes: The Web changes all the time. Furthermore, as already mentioned, users are often directly interested in changes. So, they would like a paradigm that allows to discuss change, and yes, this brings us back to the notions of temporal queries, continuous queries and subscription queries (infinite computations for the last two). So the new name of the game is infinite computation in a changing world vs. finite computation in a static one.

A world of uncertainly and incompleteness: By the nature of the Web, the information that can be acquired is incomplete and cannot be completely trusted (e.g. dangling pointers, changing or disappearing data). Query languages have to deal with this. (See, e.g., [4].)

Concurrency control: A major achievement of database technology has been concurrency control ensuring *correct* simultaneous interaction with the database by multiple users. This works fine in a centralized database with locks. It is still an elusive goal in the context of the Web. There is a need to develop more flexible notions of correctness and the corresponding theory.

Acknowledgments: I wish to thank T. Milo, L. Segoufin, D. Suciu, P. Veltri and, in particular, V. Vianu, for discussions on this paper.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan-Kaufman, New York, 1999.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading-Massachusetts, 1995.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1, 1997.

[4] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. In *Proc. ACM PODS*, 2001.

[5] S. Abiteboul and V. Vianu. Querying the Web. In *Proc. ICDT*, 1997.

[6] J. Berstel and L. Boasson. XML grammars. In *Proceedings of MFCS*, 2000.

[7] R. G. Cattell. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[8] Dblp bibliography. `www.informatik.uni-trier.de:80/ ley/db/`.

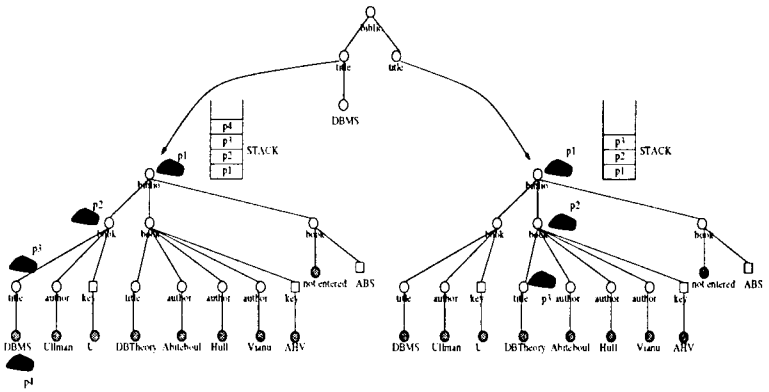


Figure 5. K-Pebble Automata Configuration

[9] T. Milo, D. Suciu, and V. Vianu. Type-checking for XML transducers. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 2000.

[10] F. Neven and T. Schwentick. Query automata. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1999.

[11] P.Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proc. DBPL*, 1995.

[12] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases (DOOD)*, Singapore, 1995. Springer-Verlag.

[13] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[14] D. Suciu. Semistructured data tutorial. VLDB99, http://www.cs.washington.edu/homes/suciu/, 1999.

[15] J.D. Ullman. *Principles of Database and Knowledge Base Systems, Volume I.* Computer Science Press, 1988.

[16] J.D. Ullman. *Principles of Database and Knowledge Base Systems, Volume II: The New Technologies.* Computer Science Press, 1989.

[17] V. Vianu. A Web odyssey: from Codd to XML. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 2001.

[18] The World Wide Web Consortium (W3C). www.w3.org.

[19] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[20] Xyleme Home Page. www.xyleme.com.

Session 10

Synthesizing Distributed Systems

Orna Kupferman
Hebrew University*

Moshe Y. Vardi
Rice University†

Abstract

In system synthesis, we transform a specification into a system that is guaranteed to satisfy the specification. When the system is distributed, the goal is to construct the system's underlying processes. Results on multi-player games imply that the synthesis problem for linear specifications is undecidable for general architectures, and is nonelementary decidable for hierarchical architectures, where the processes are linearly ordered and information among them flows in one direction. In this paper we present a significant extension of this result. We handle both linear and branching specifications, and we show that a sufficient condition for decidability of the synthesis problem is a linear or cyclic order among the processes, in which information flows in either one or both directions. We also allow the processes to have internal hidden variables, and we consider communications with and without delay. Many practical applications fall into this class.

1 Introduction

In *system synthesis*, we transform a specification into a system that is guaranteed to satisfy the specification. Early work on synthesis consider *closed systems*. There, a system that meets the specification can be extracted from a constructive proof that the specification is satisfiable [MW80, EC82]. As argued in [ALW89, Dil89, PR89a], such synthesis paradigms are not of much interest when applied to *open systems*, which interact with an environment. While synthesis that is based on satisfiability assumes no environment or a cooperative one, synthesis of open systems should assume a hostile environment, and should generate a system that satisfies the specification no matter how the environment behaves. The work in [ALW89, PR89a] formulated the synthesis problem in terms of a *game* between the system and the environment, and is closely related to *Church's solvability problem* [Chu63]. Given sets I and O of input and output signals, respectively, we can view a system as a *strategy* $P : (2^I)^* \to 2^O$ that maps a finite sequence of sets of input signals (the behavior of the environment so far) into a set of output signals (the reaction of the system to this behavior).

When P interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $2^{I \cup O}$. We say that a specification ψ is *realizable* iff there is a strategy all of whose computations satisfy ψ, in case ψ is a linear specification, or a strategy whose induced computation tree satisfies ψ, in case ψ is a branching specification. *Synthesis* of ψ then amounts to constructing such a strategy. Solutions for the realizability and synthesis problems for specifications in the linear temporal logic LTL are presented in [ALW89, PR89a]. The solutions are extended in [PR89b, Var95] to asynchronous systems and in [KV99] to systems with incomplete information and specifications in the branching temporal logic CTL*. Methods developed for synthesis of open systems are applicable also for *supervisory control*, where instead of hostile environments we consider collaborative controllers of nondeterministic systems [RW89].

While the transition to open systems has significantly broaden the scope of synthesis to real-life designs, it is still limited to settings in which the open system consists of a single process. In a more realistic setting, that of a *distributed system*, the input to the synthesis problem consists of both the specification and an *architecture*, which may consist of more than one process and describes the communication channels between the different processes. More formally, we assume a setting with n processes, with process i referring to sets I_i, O_i, and H_i, of input, output, and hidden (internal) signals (input signals may be *external*; i.e., generated by the environment), and we want to construct for each process a strat-

*Work partially supported by BSF grant 9800096. Address: School of Computer Science and Engineering, Jerusalem 91904, Israel. Email: orna@cs.huji.ac.il

†Work partially supported by NSF grants CCR-9700061 and CCR-9988322, BSF grant 9800096, and a grant from the Intel Corporation. Address: Department of Computer Science, Houston, TX 77251-1892, U.S.A. Email: vardi@cs.rice.edu

0-7695-1281-X/01 $10.00 © 2001 IEEE

egy $P_i : (2^{I_i})^* \to 2^{O_i \cup H_i}$ so that the composition of the strategies satisfies the specification. The architecture is given by a set of conditions like $O_2 \cup O_4 \subseteq I_3$ ("the only channels to P_3 are from P_2 to P_4"). The exact definition of the composition of the strategies then depends on assumptions on the communication (e.g., whether communication involves a delay). If, for example, we want to synthesize five dining philosophers [Dij72], we can specify in temporal logic the mutual exclusion and non-starvation requirements for the philosophers, specify a two-way ring with five processes, and ask the synthesis procedure to construct appropriate strategies for the processes. Clearly, a solution for the dining philosophers that refers to a single process is not of much interest.

There are two possible ways to approach the synthesis problem for distributed systems. One approach is to use a synthesis procedure for a single process, and then *decompose* the process according to the given architecture [EC82, MW84]. While this approach has a computational advantage, known decomposition algorithms are not complete in the sense that a specification may be realizable with respect to a given architecture yet the decomposition algorithm would fail [PR90]. Thus, one can view decomposition as a heuristic for the synthesis problem, which is not guaranteed to work. The second approach is to refer to the architecture of the distributed system from the outset and construct the underlying processes directly [PR90].

Results on multi-player games imply that the realizability problem for general distributed systems is undecidable [PR79, PR90] (the results in [PR79] refer to multiple-person alternating Turing machines and are extended in [PR90] to the synthesis setting). Essentially, there is an architecture Ω (in fact, a very simple architecture, consisting of two independent processes P_1 and P_2 that interact with the same environment; that is $I_1 \cap (O_2 \cup H_2) = \emptyset$ and $I_2 \cap (O_1 \cup H_1) = \emptyset$) such that for every deterministic Turing machine M, there is an LTL formula ψ_M such that M halts on the empty tape iff ψ_M is realizable in Ω. The reduction is heavily based on P_1 and P_2 being independent, and it fails, for example, if we assume that P_2 gets its input from P_1 (i.e., $O_1 \subseteq I_2$). Indeed, it is shown in [PR79, PR90] that once we consider *hierarchical architectures*, in which the processes are linearly ordered and information flows in one direction, the realizability problem is nonelementary decidable for specifications in LTL.

The decidability result in [PR90] suffers from two limitations. First, when we synthesize a system from an LTL specification ψ, we require ψ to hold in all the computations of the system. Consequently, we cannot impose possibility requirements on the system (cf. [DTV99]). In the dining-philosophers example, while we can specify in LTL mutual exclusion, we cannot specify deadlock freedom (every finite interaction *can be* extended so that a philosopher eventually eats). In order to express possibility properties, we should specify the system using *branching temporal logic*, which enables both universal and existential path quantification [EH86, Eme90]. Second, and more crucially, the algorithm in [PR90] is not applicable for architectures that are not hierarchical, and real-life designs are rarely based on hierarchical architectures. We do not count the nonelementary complexity as a limitation, as it is accompanied by a matching lower bound and, as we discuss further in Section 6, the worst-case complexity rarely appears in practice.

In this paper we remove both limitations. We consider specifications in the branching temporal logic CTL* (which subsumes LTL), and we handle all architectures in which there is a linear or cyclic order among the processes, in which information flows in either one or both directions. Thus, our architectures can be either chains or rings with both one-way and two-way communication channels. In addition, we allow the processes to have internal hidden variables, and we consider communications with and without delay. We show that the realizability problem stays decidable in all these cases. The solution we present is based on *alternating tree automata*, which separate the logical and algorithmic aspects of the problem: given a specification ψ and an architecture Ω, we construct an automaton $\mathcal{A}_{\Omega,\psi}$ such that ψ is realizable in Ω iff $\mathcal{A}_{\Omega,\psi}$ is not empty. To check realizability, the automaton has to be tested for nonemptiness [EJ88, PR89a, KV98]. The nonemptiness algorithm also synthesizes the processes in Ω that together realize ψ.

We argue that the results in the paper significantly extend the scope of synthesis for distributed systems, as commonly used architecture belong to the class of architectures we handle [Tan87]. Examples of applications of these architectures include various communication protocols in which communication proceeds in layers. For example, the so-called OSI model consists of a seven-layer *protocol stack* (Application, Presentation, Session, Transport, Network, Data link, and Physical layers), where every layer communicates with the layer above it and the layer below it. The environment talks to the top layer and the bottom layer [Man99]. Architectures with two-way communication channels are common in *scientific computations*, say when we iterate in order to solve a differential equa-

tion and each process works on part of the computed domain. Then, it is useful to divide the domain to layers so that in each iteration every layer updates its neighbors with its results from the previous iteration [PTVF92].

2 Preliminaries

2.1 Trees and labeled trees

Given a finite set Υ, an Υ-*tree* is a set $T \subseteq \Upsilon^*$ such that if $x \cdot \upsilon \in T$, where $x \in \Upsilon^*$ and $\upsilon \in \Upsilon$, then also $x \in T$. When Υ is not important or clear from the context, we call T a tree. When $T = \Upsilon^*$, we say that T is *full*. The elements of T are called *nodes*, and the empty word ϵ is the *root* of T. For every $x \in T$, the nodes $x \cdot \upsilon \in T$ where $\upsilon \in \Upsilon$ are the *children* of x. Each node x of T has a *direction*, $dir(x)$ in Υ. The direction of ϵ is υ^0, for some designated $\upsilon^0 \in \Upsilon$, called the *root direction*. The direction of a node $x \cdot \upsilon$ is υ.

Given two finite sets Υ and Σ, a Σ-*labeled* Υ-*tree* is a pair $\langle T, V\rangle$ where T is an Υ-tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, V\rangle$ a labeled tree. For a Σ-labeled Υ-tree $\langle \Upsilon^*, V\rangle$, we define the *memoryfull* version of $\langle \Upsilon^*, V\rangle$, denoted $mem(\langle \Upsilon^*, V\rangle)$ as the Σ^+-labeled Υ-tree $\langle \Upsilon^*, V'\rangle$ where $V'(\epsilon) = V(\epsilon)$, for $\upsilon \in \Upsilon$ we have $V'(\upsilon) = V(\epsilon) \cdot V(\upsilon)$, and for all $x \in \Upsilon^+$ and $\upsilon \in \Upsilon$ we have $V'(x \cdot \upsilon) = V'(x) \cdot V(\upsilon)$. Thus, the label of a node x in $mem(\langle \Upsilon^*, V\rangle)$ is the word obtained by concatenating the labels of all the prefixes (including ϵ) of x in $\langle \Upsilon^*, V\rangle$.

For a Σ-labeled Υ-tree $\langle \Upsilon^*, V\rangle$, we define the *x-ray* of $\langle \Upsilon^*, V\rangle$, denoted $xray(\langle \Upsilon^*, V\rangle)$, as the $(\Upsilon \times \Sigma)$-labeled Υ-tree $\langle \Upsilon^*, V'\rangle$ in which each node is labeled by both its direction and its labeling in $\langle \Upsilon^*, V\rangle$. Thus, for every $x \in \Upsilon^*$, we have $V'(x) = \langle dir(x), V(x)\rangle$. Essentially, the labels in $xray(\langle \Upsilon^*, V\rangle)$ contain information not only about the surface of $\langle \Upsilon^*, V\rangle$ (its labels) but also about its skeleton (its nodes).

For a Σ-labeled Υ-tree $\langle \Upsilon^*, V\rangle$, we define the *delay* of $\langle \Upsilon^*, V\rangle$, denoted $delay(\langle \Upsilon^*, V\rangle)$, as the Σ-labeled Υ-tree $\langle \Upsilon^*, V'\rangle$ in which $V'(\epsilon) = V(\epsilon)$ and for all $x \in \Upsilon^*$ and $\upsilon \in \Upsilon$, we have $V'(x \cdot \upsilon) = V(\upsilon_0 \cdot x)$, where $\upsilon_0 = dir(\epsilon)$ is the root direction of Υ. Intuitively, the delay of $\langle \Upsilon^*, V\rangle$ describes the label node x would have when the sequence of directions leading to x arrives with a delay, thus the last direction in x is missing and x is prefixed by the root direction.

Consider a set $X \times Y$ of directions. For a node $\tau \in (X \times Y)^*$, let $hide_Y(\tau)$ be the node in X^* obtained from τ by replacing each letter $\langle x, y\rangle$ by the letter x. For example, the node $\langle 0,0\rangle \cdot \langle 1,0\rangle$ of the 4-ary $(\{0,1\} \times \{0,1\})$-tree corresponds, by $hide_{\{0,1\}}$, to the node $0 \cdot 1$ of the $\{0,1\}$-tree. Note that the nodes $\langle 0,0\rangle \cdot \langle 1,1\rangle$, $\langle 0,1\rangle \cdot \langle 1,0\rangle$, and $\langle 0,1\rangle \cdot \langle 1,1\rangle$ of the 4-ary tree also correspond, by $hide_{\{0,1\}}$, to the node $0 \cdot 1$ of the binary tree. For a Z-labeled X-tree $\langle X^*, V\rangle$, we define the Y-*widening* of $\langle X^*, V\rangle$, denoted $wide_Y(\langle X^*, V\rangle)$, as the Z-labeled $(X \times Y)$-tree $\langle (X \times Y)^*, V'\rangle$ where for every $\tau \in (X \times Y)^*$, we have $V'(\tau) = V(hide_Y(\tau))$. As we explain further in Section 3, nodes τ_1 and τ_2 with $hide_Y(\tau_1) = hide_Y(\tau_2) = \tau$ are indistinguishable in $wide_Y(\langle X^*, V\rangle)$ by someone that does not observe Y. Indeed, for such an observer, both nodes are reached by traversing τ and are labeled by $V(\tau)$.

2.2 Alternating automata

Alternating tree automata generalize nondeterministic tree automata and were first introduced in [MS87]. An alternating tree automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha\rangle$ runs on full Σ-labeled Υ-trees (for an agreed set Υ of directions). It consists of a finite set Q of states, an initial state $q_0 \in Q$, a transition function δ, and an acceptance condition α (a condition that defines a subset of Q^ω). For a set Υ of directions, let $\mathcal{B}^+(\Upsilon \times Q)$ be the set of positive Boolean formulas over $\Upsilon \times Q$; i.e., Boolean formulas built from elements in $\Upsilon \times Q$ using $\wedge$ and $\vee$, where we also allow the formulas **true** and **false** and, as usual, $\wedge$ has precedence over $\vee$. The transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(\Upsilon \times Q)$ maps a state and an input letter to a formula that suggests a new configuration for the automaton. For example, when $\Upsilon = \{0,1\}$, having $\delta(q, \sigma) = (0, q_1) \wedge (0, q_2) \vee (0, q_2) \wedge (1, q_2) \wedge (1, q_3)$ means that when the automaton is in state q and reads the letter σ, it can either send two copies, in states q_1 and q_2, to direction 0 of the tree, or send a copy in state q_2 to direction 0 and two copies, in states q_2 and q_3, to direction 1. Thus, unlike nondeterministic tree automata, here the transition function may require the automaton to send several copies to the same direction or allow it not to send copies to all directions.

A *run* of an alternating automaton $\mathcal{A}$ on an input Σ-labeled Υ-tree $\langle T, V\rangle$ is a tree $\langle T_r, r\rangle$ in which the nodes are labeled by elements of $\Upsilon^* \times Q$. Each node of T_r corresponds to a node of T. A node in T_r, labeled by (x, q), describes a copy of the automaton that reads the node x of T and visits the state q. Note that many nodes of T_r can correspond to the same node of T; in contrast, in a run of a nondeterministic automaton on $\langle T, V\rangle$ there is a one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its

children have to satisfy the transition function. For example, if $\langle T, V\rangle$ is a $\{0,1\}$-tree with $V(\epsilon) = a$ and $\delta(q_0, a) = ((0, q_1) \vee (0, q_2)) \wedge ((0, q_3) \vee (1, q_2))$, then the nodes of $\langle T_r, r\rangle$ at level 1 include the label $(0, q_1)$ or $(0, q_2)$, and include the label $(0, q_3)$ or $(1, q_2)$. Each infinite path ρ in $\langle T_r, r\rangle$ is labeled by a word $r(\rho)$ in Q^ω. Let $inf(\rho)$ denote the set of states in Q that appear in $r(\rho)$ infinitely often. A run $\langle T_r, r\rangle$ is accepting iff all its infinite paths satisfy the acceptance condition. In *Rabin* alternating tree automata, $\alpha \subseteq 2^Q \times 2^Q$, and an infinite path ρ satisfies an acceptance condition $\alpha = \{\langle G_1, B_1\rangle, \ldots, \langle G_k, B_k\rangle\}$ iff there exists $1 \leq i \leq k$ for which $inf(\rho) \cap G_i \neq \emptyset$ and $inf(\rho) \cap B_i = \emptyset$. We refer to the number of pairs in α as the *index* of $\mathcal{A}$. An automaton accepts a tree iff there exists an accepting run on it. We denote by $\mathcal{L}(\mathcal{A})$ the language of the automaton $\mathcal{A}$; i.e., the set of all labeled trees that $\mathcal{A}$ accepts. We say that an automaton is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$. For an acceptance condition α over Q and a set S, we denote by $\alpha \times S$ the acceptance condition over $Q \times S$ obtained from α by replacing each set F participating in α by the set $F \times S$. For example, if α is the Rabin acceptance condition $\{\langle G, B\rangle\}$, then $\alpha \times S = \{\langle G \times S, B \times S\rangle\}$.

Nondeterministic tree automata can be viewed as a special case of alternating tree automata, where the formulas in $\mathcal{B}^+(\Upsilon \times Q)$ are such that if a formula is rewritten in disjunctive normal form, then for every direction $v \in \Upsilon$, there is exactly one element of $\{v\} \times Q$ in each disjunct. While nondeterministic tree automata are not less expressive than alternating tree automata, they are exponentially less succinct:

Theorem 2.1 [MS95] *An alternating Rabin tree automaton with m states and k pairs can be translated to an equivalent nondeterministic Rabin tree automaton with $m^{O(mk)}$ states and $O(mk)$ pairs.*

3 Architectures and the synthesis problem

Given sets I and O of input and output signals, respectively, we can view a process P as a *strategy* $f : (2^I)^* \to 2^O$ that maps a finite sequence of sets of input signals into a set of output signals. We often refer to the strategy f as the 2^O-labeled 2^I-tree $\langle (2^I)^*, f\rangle$. Let i_0 be the root direction of 2^I. When P interacts with an environment that generates infinite input sequences, it associates with each infinite input sequence $i_1, i_2, \ldots$, an infinite computation $\{i_0\} \cup f(\varepsilon), \{i_1\} \cup f(i_1), \{i_2\} \cup f(i_1 \cdot i_2), \ldots$ over $2^{I \cup O}$. The interaction of P with all possible input sequences induces the $(2^{I \cup O})$-labeled 2^I-tree $xray(\langle (2^I)^*, f\rangle)$. The environment may have hidden internal signals, which are not readable by P. Let H denote the set of hidden signals. Then, a strategy for P is still a function $f : (2^I)^* \to 2^O$, but the interaction of P with an outcome of the environment induces an infinite computation over $2^{I \cup O \cup H}$, and its interaction with all possible outcomes induces the $(2^{I \cup O \cup H})$-labeled $(2^{I \cup H})$-tree $xray(wide_{(2^H)}(\langle (2^I)^*, f\rangle))$. Each node in this tree has $2^{|I \cup H|}$ children[1], corresponding to the $2^{|I \cup H|}$ possible assignments to $I \cup H$. Note that since P cannot see the signals in H, and thus cannot distinguish between children that agree on their assignment to signals in I, the tree above is the 2^H-widening of the interaction between P and its environment as seen by P.

In a setting with n processes $P_1, \ldots, P_n$, where process P_i reads I_i, writes O_i, and has hidden internal signals H_i, a strategy for P_i is a function $f_i : (2^{I_i})^* \to 2^{O_i \cup H_i}$. We denote $\bigcup_{1 \leq i \leq n} I_i$ by I, and similarly for O and H. The n processes $P_1, \ldots, P_n$ interact with each other and may also interact with an environment. We denote by O_{env} the output signals of the environment (that is, the external input to the n processes), and denote by H_{env} the hidden signals of the environment.

Different architectures induce different communication channels between the processes. We consider here four classes of architectures (see figure next page). In all classes, each signal can be written by a single process (that is, $O_i \cap O_j = \emptyset$ for all $i \neq j$), but can be read by several processes (that is, possibly $I_i \cap I_j \neq \emptyset$).

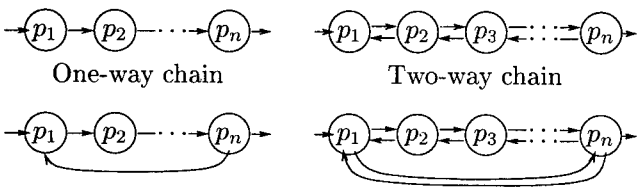
- In a *one-way chain*, P_1 reads from the environment, P_n writes to the environment, and all the other processes read from the process to their left, and write to the process to their right. Formally, $I_1 = O_{env}$, and for all $2 \leq i \leq n$ we have $I_i = O_{i-1}$. Note that P_i cannot read the internal signals of the process to its left and that $I \cup O = I \cup O_n = I_1 \cup O$.
- A *one-way ring* extends a one-way chain by a communication channel from P_n to P_1. Thus, P_1 reads from both P_n and the environment (i.e., $I_1 = O_n \cup O_{env}$), and P_n writes to both P_1 and the environment.
- In a *two-way chain*, P_1 reads from both P_2 and the environment and writes to P_2, P_n reads from P_{n-1} and writes to both P_{n-1} and the environment, and all the other processes read from the

[1] We consider synthesis with respect to *maximal environments*, which provide all possible input sequences. An extension to non-maximal environment is possible, using the same techniques as in [KMTV00].

processes to their left and right, and write to the processes to their left and right. Formally, $I_1 = O_{env} \cup O_2$, for all $2 \leq i \leq n-1$ we have $I_i = O_{i-1} \cup O_{i+1}$, and $I_n = O_{n-1}$.

- A *two-way ring* extends a two-way chain by a communication channel between P_n and P_1. Thus, P_1 reads from P_2, P_n, and the environment (i.e., $I_1 = O_{env} \cup O_2 \cup O_n$), and writes to both P_2 and P_n, and P_n reads from both P_1 and P_{n-1} and writes to both P_1, P_{n-1}, and the environment.

Note that in all the four classes, and for all i and j with $i < j$, the process P_i has complete information about the input to P_j, thus P_i can simulate P_j and have complete information also about its output[2]. This means, for example, that in a two-way chain, we could give up the channel from P_2 to P_1, letting P_1 compute the information along this channel, and similarly for the other right-to-left channels. While this would not change the answer to the realizability question, it may significantly increase the sizes of the synthesized processes.



For all the architectures, we define the *composition* of strategies $f_1, \ldots, f_n$ as a function $f : (2^{O_{env}})^* \to 2^{O \cup H}$ that describes the joint behavior of the processes on an infinite sequence of external input signals. The exact definition of a composition depends on the particular architecture as well as on assumptions on the communication (e.g., whether communication involves a delay). We define several compositions in Section 5. In [PR90], Pnueli and Rosner study one-way channels (called "hierarchical architectures" there) where communication involves no delay. In this setting, compositions are defined as follows. For the strategy $\langle (2^{I_i})^*, f_i \rangle$, let $\langle (2^{I_i})^*, f_i' \rangle = mem(\langle (2^{I_i})^*, f_i \rangle)$. Recall that in a one-way chain, $O_{env} = I_1$. Then, $f : (2^{O_{env}})^* \to 2^{O \cup H}$ is such that for every $\sigma \in (2^{O_{env}})^*$, we have

$$f(\sigma) = f_1(\sigma) \cup f_2(f_1'(\sigma)) \cup f_3(f_2'(f_1'(\sigma))) \cup \ldots \cup f_n(f_{n-1}'(\cdots(f_2'(f_1'(\sigma)))\cdots)).$$

Intuitively, for all i, the output of P_i (and, consequently, the contribution of f_i to f), depends on the history of the outputs of P_{i-1}, namely the memoryfull version of f_{i-1}, which by itself depends on the memoryfull version of f_{i-2}, and so on.

The composition f induces the *computation tree* of $P_1, \ldots, P_n$, which is the $(2^{I \cup O \cup H \cup H_{env}})$-labeled $(2^{O_{env} \cup H_{env}})$-tree $xray(wide_{(2^{H_{env}})}(\langle (2^{O_{env}})^*, f \rangle))$. The transition from the composition to the computation tree involves two transformations. First, while the composition f corresponds to the composition as seen by the processes, and thus ignores the signals in H_{env} and the nondeterminism induced by them, the computation tree corresponds to the composition as seen by someone that sees all signals, which involves a $2^{H_{env}}$-widening. In addition, as the signals in O_{env} and H_{env} are represented in the widening of the composition only in its nodes and not in its labels, we employ *xray* and obtain a tree whose labels refer to all signals.

Given a CTL* formula ψ over $I \cup O \cup H \cup H_{env}$, and an architecture Ω with processes $P_1, \ldots, P_n$, we say that ψ is *realizable* in Ω iff there are strategies for $P_1, \ldots, P_n$ whose composition induces a computation tree that satisfies ψ. The *synthesis problem* is then to construct these strategies. The synthesis problem for one-way chains with complete information is introduced and solved in [PR90] for specifications in the linear temporal logic LTL (which is a strict subset of CTL*). The synthesis problem for CTL* for an architecture with a single process with incomplete information is introduced and solved in [KV99]. In this paper, we solve the synthesis problem for CTL* for the four classes of architectures introduced above. Our solution is based on automata on infinite trees. For our purposes, the crucial feature of CTL* is the following translation of CTL* formulas to alternating Rabin tree automata.

Theorem 3.1 [KVW00] *Given a CTL* formula ψ over a set AP of atomic propositions and a set Υ of directions, there exists an alternating Rabin tree automaton $\mathcal{A}_{\Upsilon,\psi}$ over 2^{AP}-labeled Υ-trees, with $2^{O(|\psi|)}$ states and two pairs, such that $\mathcal{L}(\mathcal{A}_{\Upsilon,\psi})$ is exactly the set of trees satisfying ψ.*

4 Useful automata constructions

Let X, Y, and Z be finite sets, and let z_0 be the root direction of Z. For an $(X \times Y)$-labeled Z-tree $\langle Z^*, f \rangle$,

[2] Indeed P_j, for $j > i$, generates also hidden signals, but these signals are generated by a strategy that is known to P_i, since our framework assumes that the processes are collaborative, while the environment is adversarial.

we say that $\langle Z^*, f\rangle$ is a *composition* of an X-labeled Z-tree $\langle Z^*, f_X\rangle$, where $mem(\langle Z^*, f_X\rangle) = \langle Z^*, f'_X\rangle$, and a Y-labeled X-tree $\langle X^*, f_Y\rangle$ iff for every z_1 and z_2 in Z and for every $\sigma \in Z^*$, we have

- $f(\epsilon) = f_X(\epsilon) \cup f_Y(\epsilon)$.
- $f(z_1) = f_X(z_0) \cup f_Y(f'_X(\epsilon))$.
- $f(\sigma \cdot z_1 \cdot z_2) = f_X(z_0 \cdot \sigma \cdot z_1) \cup f_Y(f'_X(z_0 \cdot \sigma))$.

We then say that $f = f_X + f_Y$. For a set $\mathcal{T}$ of $(X \times Y)$-labeled Z-trees, the set $shape_X(\mathcal{T})$ consists of all Y-labeled X-trees $\langle X^*, f_Y\rangle$ for which there exists an X-labeled Z-tree $\langle Z^*, f_X\rangle$ such that the $(X \times Y)$-labeled Z-tree $\langle Z^*, f_X + f_Y\rangle$ is in $\mathcal{T}$.

Theorem 4.1 *Let X, Y, and Z be finite sets. Given a nondeterministic tree automaton $\mathcal{A}$ over $(X \times Y)$-labeled Z-trees, we can construct an alternating tree automaton $\mathcal{A}'$ over Y-labeled X-trees such that $\mathcal{L}(\mathcal{A}') = shape_X(\mathcal{L}(\mathcal{A}))$ and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Proof: Let $\mathcal{A} = \langle X \times Y, Q, q_0, \delta, \alpha\rangle$. Then, $\mathcal{A}' = \langle Y, Q, q_0, \delta', \alpha\rangle$, where for every $q \in Q$ and $y \in Y$, we have

$$\delta'(q, y) = \bigvee_{\substack{x \in X, \\ \langle s_1, s_2, \ldots, s_{|Z|}\rangle \in \delta(q, \langle x, y\rangle)}} (x, s_1) \wedge (x, s_2) \wedge \ldots \wedge (x, s_{|Z|}).$$

Consider first the case where $q = q_0$ and $\mathcal{A}'$ reads the root of the input tree $\langle X^*, f_Y\rangle$. The letter y read at the root is $f_Y(\epsilon)$. Since in $f_X + f_Y$ the root is labeled $\langle f_X(\epsilon), f_Y(\epsilon)\rangle$, we proceed according to $\delta(q_0, \langle x, y\rangle)$ for some x which is our guess for $f_X(\epsilon)$. By the definition of δ', each copy of $\mathcal{A}$ that is sent to direction $z \in Z$ and visits state s induces a copy of $\mathcal{A}'$ that is sent to direction x and visits the state s. Since the choice of x is joint to all $z \in Z$, all the copies of $\mathcal{A}'$ induced as above are going to read the same letter, which is our guess for $f_Y(f_X(\epsilon))$. Consider now a copy of $\mathcal{A}$ that reads a node $z \in Z$ and visits state s. Recall that the automaton $\mathcal{A}'$ then has a copy that reads the node $f_X(\epsilon)$, visits the state s, and the letter y read by this copy (and all the other copies that read the node $f_X(\epsilon)$) is our guess for $f_Y(f_X(\epsilon))$. Since in $f_X + f_Y$ the node z is labeled $\langle f_X(z_0), f_Y(f_X(\epsilon))\rangle$, we proceed according to $\delta(s, \langle x, y\rangle)$, for some x which is our guess for $f_X(z_0)$. Each copy of $\mathcal{A}$ that is sent to direction $z' \in Z$ and visits state s' then induces a copy of $\mathcal{A}'$ that is sent to direction x and visits the state s'. All these copies are going to read the same letter, which is our guess for $f_Y(f'_X(z_0))$. The same idea repeats in further levels: a copy of $\mathcal{A}$ that reads a node $\sigma \cdot z_1 \cdot z_2 \in Z^*$ and visits state s is associated with a copy of $\mathcal{A}'$ that reads the node $f'_X(z_0 \cdot \sigma)$ and visits the state s. The letter y read by this copy (and all the other copies that read the node $f'_X(z_0 \cdot \sigma)$) is our guess for $f_Y(f'_X(z_0 \cdot \sigma))$. Since in $f_X + f_Y$ the node $\sigma \cdot z_1 \cdot z_2$ is labeled $\langle f_X(z_0 \cdot \sigma \cdot z_1), f_Y(f'_X(z_0 \cdot \sigma))\rangle$, we proceed according to $\delta(s, \langle x, y\rangle)$ for some x which is our guess for $f_X(z_0 \cdot \sigma \cdot z_1)$. All the copies sent to direction x are going to read the same letter, which is our guess for $f_Y(f'_X(z_0 \cdot \sigma \cdot z_1))$. □

Given a nondeterministic tree automaton $\mathcal{A}$, let $shape_X(\mathcal{A})$ denote the corresponding automaton $\mathcal{A}'$ constructed in Theorem 4.1. Note that while $shape_X(\mathcal{A})$ returns an alternating tree automaton, it is defined for a nondeterministic tree automaton $\mathcal{A}$. Thus, successive applications of *shape* require an intermediate application of the exponential alternation-removal procedure in Theorem 2.1.

The construction described in Theorem 4.1 will help us to solve the realizability problem by successively reducing the number of processes in the architectures. The two constructions below will handle the external input to the system and the incomplete information, and they are presented in [KV99], where they are used for the synthesis of a single process with incomplete information.

Theorem 4.2 *Given an alternating tree automaton $\mathcal{A}$ over $(\Upsilon \times \Sigma)$-labeled Υ-trees, we can construct an alternating tree automaton $\mathcal{A}'$ over Σ-labeled Υ-trees such that $\mathcal{A}'$ accepts a labeled tree $\langle \Upsilon^*, V\rangle$ iff $\mathcal{A}$ accepts $xray(\langle \Upsilon^*, V\rangle)$, and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Theorem 4.3 *Let X, Y, and Z be finite sets. Given an alternating tree automaton $\mathcal{A}$ over Z-labeled $(X \times Y)$-trees, we can construct an alternating tree automaton $\mathcal{A}'$ over Z-labeled X-trees such that $\mathcal{A}'$ accepts a Z-labeled tree $\langle X^*, V\rangle$ iff $\mathcal{A}$ accepts the Z-labeled tree $wide_Y(\langle X^*, V\rangle)$, and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Finally, since we want our algorithm to be applicable also for settings in which communication involves a delay, we need a construction that handles such a delay.

Theorem 4.4 *Given an alternating tree automaton $\mathcal{A}$ over Σ-labeled Υ-trees, we can construct an alternating tree automaton $\mathcal{A}'$ over Σ-labeled Υ-trees such that $\mathcal{A}'$ accepts a labeled tree $\langle \Upsilon^*, V\rangle$ iff $\mathcal{A}$ accepts $delay(\langle \Upsilon^*, V\rangle)$, and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Given an alternating tree automaton $\mathcal{A}$, let $cover(\mathcal{A})$, $narrow_Y(\mathcal{A})$, and $wait(\mathcal{A})$ denote the corresponding automata $\mathcal{A}'$ constructed in Theorems 4.2, 4.3 (for a set Y of directions), and 4.4, respectively.

5 Solving the synthesis problem

In this section we study the synthesis problem for the architectures described in Section 3. We show that for all the four classes, the problem is decidable, with a nonelementary complexity. Thus, given a CTL* formula ψ, a class $\mathcal{C}$ (one-way chain, two-way chain, one-way ring, or two-way ring), and an integer n, the complexity of constructing n strategies for n processes in an architecture of class $\mathcal{C}$ that satisfies ψ is $n\text{-}exp(|\psi|)$.[3]

One-way chain We assume that communication involves a delay. Thus, the input to P_{i+1} at time t is the output of P_i (or the environment, when $i = 0$) at time $t-1$. Accordingly, we define the *composition* f of $f_1, \ldots, f_n$ as follows. For a string $\sigma = z_0 \cdot z_1 \cdots z_k$ and $i \geq 0$, let $z_0 \cdot z_1 \cdots z_{k-i}$ be either the prefix of length $k-i+1$ of σ, in case $k-i \geq 0$, or ϵ, in case $k-i+1 \leq 0$. Also, let z_0 be the root direction of 2^{I_1}. Then, $f : (2^{I_1})^* \to 2^{O \cup H}$ is defined as follows.

- $f(\epsilon) = f_1(\epsilon) \cup \cdots \cup f_n(\epsilon)$.
- For $\sigma \in (2^{I_1})^*$ with $\sigma = z_1 \cdots z_k$, we have $f(\sigma) = f_1(z_0 \cdot z_1 \cdots z_{k-1}) \cup f_2(f'_1(z_0 \cdot z_1 \cdots z_{k-2})) \cup \cdots \cup f_n(f'_{n-1}(z_0 \cdot z_1 \cdots z_{k-n}))$.

Consider a CTL* formula ψ over $I \cup O \cup H \cup H_{env}$. Recall that in a one-way chain, we have $I \cup O = I_1 \cup O$. In order to solve the realizability problem, we build the following tree automata.

- $\mathcal{A}_\psi$: an alternating Rabin tree automaton that accepts a $(2^{I_1 \cup O \cup H \cup H_{env}})$-labeled $(2^{I_1 \cup H_{env}})$-tree $\langle (2^{I_1 \cup H_{env}})^*, f \rangle$ iff it satisfies ψ [see Theorem 3.1].
- $\mathcal{A}_0$: the alternating Rabin tree automaton $wait(\mathcal{A}_\psi)$. Thus, $\mathcal{A}_0$ accepts a $(2^{I_1 \cup O \cup H \cup H_{env}})$-labeled $(2^{I_1 \cup H_{env}})$-tree $\langle (2^{I_1 \cup H_{env}})^*, f \rangle$ iff $delay(\langle (2^{I_1 \cup H_{env}})^*, f \rangle)$ satisfies ψ [see Theorem 4.4].
- $\mathcal{A}'_0$: the alternating Rabin tree automaton $cover(\mathcal{A}_0)$. Thus, $\mathcal{A}'_0$ accepts a $(2^{O \cup H})$-labeled $(2^{I_1 \cup H_{env}})$-tree $\langle (2^{I_1 \cup H_{env}})^*, f \rangle$ iff $delay(xray(\langle (2^{I_1 \cup H_{env}})^*, f \rangle))$ satisfies ψ [see Theorem 4.2].
- $\mathcal{A}''_0$: the alternating Rabin tree automaton $narrow_{(2^{H_{env}})}(\mathcal{A}'_0)$. Thus, $\mathcal{A}''_0$ accepts a $(2^{O \cup H})$-labeled 2^{I_1}-tree $\langle (2^{I_1})^*, f \rangle$ iff $delay(xray(wide_{(2^{H_{env}})}(\langle (2^{I_1})^*, f \rangle)))$ satisfies ψ [see Theorem 4.3].
- For $1 \leq i \leq n-1$,
  - $\mathcal{A}_i$: a nondeterministic Rabin tree automaton equivalent to $\mathcal{A}''_{i-1}$ [see Theorem 2.1]. Note that the automaton $\mathcal{A}_i$ runs on $(2^{O_i \cup H_i \cup O_{i+1} \cup H_{i+1} \cup \cdots \cup O_n \cup H_n})$-labeled $2^{O_{i-1}}$-trees, where we take $O_0 = I_1$.
  - $\mathcal{A}'_i$: the alternating Rabin automaton $shape_{(2^{O_i \cup H_i})}(\mathcal{A}_i)$. Thus, $\mathcal{A}'_i$ runs on $(2^{O_{i+1} \cup H_{i+1} \cup \cdots \cup O_n \cup H_n})$-labeled $(2^{O_i \cup H_i})$-trees and it accepts a tree $\langle (2^{O_i \cup H_i})^*, f \rangle$ iff there is a $(2^{O_i \cup H_i})$-labeled $2^{O_{i-1}}$-tree $\langle (2^{O_{i-1}})^*, f' \rangle$ such that $\langle (2^{O_{i-1}})^*, f + f' \rangle$ is accepted by $\mathcal{A}_i$ [see Theorem 4.1].
  - $\mathcal{A}''_i$: the alternating Rabin automaton $narrow_{(2^{H_i})}(\mathcal{A}'_i)$. Thus, $\mathcal{A}''_i$ accepts a $(2^{O_{i+1} \cup H_{i+1} \cup \cdots \cup O_n \cup H_n})$-labeled 2^{O_i}-tree $\langle (2^{O_i})^*, f \rangle$ iff $wide_{(2^{H_i})}(\langle (2^{O_i})^*, f \rangle)$ is accepted by $\mathcal{A}'_i$ [see Theorem 4.3].

Intuitively, in each iteration $1 \leq i \leq n$, we assume that the strategies of $P_1, \ldots, P_{i-1}$ are given (they are encapsulated in the transition function of $\mathcal{A}_i$) and the automaton $\mathcal{A}_i$ accepts all the compositions of $P_i, \ldots P_n$ that together with the given strategies satisfy ψ. Thus, the transition from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$ involves an encapsulation of the possible strategies of P_i (and how they affect the behavior required from $P_{i+1}, \ldots, P_n$ in order to satisfy ψ) into the transition function of $\mathcal{A}_i$.

Lemma 5.1 *ψ is realizable iff $\mathcal{A}'_{n-1}$ is not empty.*

The construction of $\mathcal{A}_i$ goes via i iterations. Each iteration involves two automata transformations. One transformation (*narrow*) gets and returns an alternating tree automaton. The other transformation (*shape*) gets a nondeterministic tree automaton and return an alternating tree automaton. While all the transformations involve no blow-up in the size of the automata, the fact that *shape* handles nondeterministic automata requires the application of an additional transformation, namely the translation of an alternating tree automaton to a nondeterministic one. This transformation involves an exponential blow-up, leading to an overall nonelementary blow-up.

[3] $n\text{-}exp(k)$ is a stack of n exponents with k on the top; i.e., $1\text{-}exp(k) = 2^{O(k)}$, and $(i+1)\text{-}exp(k) = 2^{i\text{-}exp(k)}$.

Theorem 5.2 *The synthesis problem for CTL* and one-way chains is nonelementary decidable.*

Proof: It follows from the constructions described in Section 4 that the size of $\mathcal{A}''_{n-1}$ is $(n-1)$-$exp(|\psi|)$. The nonemptiness problem for $\mathcal{A}''_{n-1}$ can then be solved in time n-$exp(|\psi|)$ [MS95, KV98]. Lemma 5.1 then implies that the realizability problem for ψ can be solved in time n-$exp(|\psi|)$. The nonemptiness algorithm can be extended to produce a witness for the automaton being nonempty (in fact, a witness that is a *memoryless strategy* [Tho95]). A witness for the nonemptiness of $\mathcal{A}''_{n-1}$ induces a strategy f_n for P_n. In order to get a strategy for P_{n-1}, we combine $\mathcal{A}''_{n-2}$ with f_n and get an automaton that is guaranteed to be nonempty and whose witness induces a strategy f_{n-1} for P_{n-1}. We continue similarly until strategies for all processes are synthesized. □

A matching nonelementary lower bound is proved (for LTL formulas) in [PR90] (cf. [PR79]). This lower bounds applies also to the other architecture.

With appropriate simple modifications (skipping the "wait construction" and redefining the "shape construction" to ignore the delay), the method described above can handle one-way channels in which communication involves no delay (the definition of composition then coincides with the one of [PR90]). As we describe below, the method can also be extended to handle the other classes of architectures described in Section 3. The differences among the architectures influence the sets of labels and directions of the trees over which the automata are defined (for example, in a one-way ring $\mathcal{A}_\psi$ runs on $(2^{O_{env} \cup O_n})$-trees, and in a two-way ring, it runs on $(2^{O_{env} \cup O_2 \cup O_n})$-trees), influence the definition of composition, and accordingly influence the definition of $shape_X(\mathcal{T})$ and the "shape construction" that handles. For all the architectures, however, the idea is similar: a successive reduction in the number of processes, where in each step we omit a process and encapsulate its possible strategies into the transition function of intermediate automata.

One-way ring. Recall that in a one-way ring, the process P_1 reads signals from both P_n and the environment. We suggest two alternative modifications to the method presented for one-way chains. The first is rather simple: all the intermediate automata we construct maintain (in their alphabet) the input that P_1 reads from P_n. Then, in the last automaton, which corresponds to P_n's strategy, we close the ring by requiring the output of P_n to agree with the maintained input. The second approach is cleaner (and it also has a computational advantage), yet it requires a more substantial modification. The idea is to start with P_1 and proceed in both directions, encapsulating two processes in each iteration. The two directions meet at the automaton $\mathcal{A}_{\frac{n}{2}}$, whose nonemptiness witnesses a strategy for $P_{\frac{n}{2}}$ that satisfies the tasks inherited to $P_{\frac{n}{2}}$ by both the processes to his left and these to his right.

Two-way chain. The two-way chain architecture is much richer than that of a one-way chain. Since the difficulties imposed by incomplete information are orthogonal and are handled by the narrow construction, we describe here the solution for systems with complete information, thus $H_{env} \cup H = \emptyset$. In a two-way chain, the process P_i reads both O_{i-1} and O_{i+1}, so its strategy is a function $f_i : (2^{O_{i-1} \cup O_{i+1}})^* \to 2^{O_i}$. Accordingly, while in the case of a one-way chain the reduction of the process P_i involves a transition from an automaton that runs on $(2^{O_i \cup O_{i+1} \cup \cdots \cup O_n})$-labeled $2^{O_{i-1}}$-trees to an automaton that runs on $(2^{O_{i+1} \cup \cdots \cup O_n})$-labeled 2^{O_i}-trees, here the reduction of P_i should involve a transition from an automaton that runs on $(2^{O_i \cup O_{i+1} \cup \cdots \cup O_n})$-labeled $(2^{O_{i-1} \cup O_{i+1}})$-trees to an automaton that runs on $(2^{O_{i+1} \cup \cdots \cup O_n})$-labeled $(2^{O_i \cup O_{i+2}})$-trees. In order to see the modifications that are therefore needed in the shape construction, let us first redefine the predicate *shape* and the composition operator it involves.

Let X_{i-1}, X_i, X_{i+1}, X_{i+2}, and X be finite sets, and let z_0 and z'_0 be the root directions of X_{i-1} and X_{i+1} respectively. For our application, X_j stands for 2^{O_j}, and X stands for $2^{O_{i+3} \cup \ldots \cup O_n}$. For an $(X_i \times X_{i+1} \times X_{i+2} \times X)$-labeled $(X_{i-1} \times X_{i+1})$-tree $\langle (X_{i-1} \times X_{i+1})^*, f \rangle$, we say that $\langle (X_{i-1} \times X_{i+1})^*, f \rangle$ is a *composition* of an X_i-labeled $(X_{i-1} \times X_{i+1})$-tree $\langle (X_{i-1} \times X_{i+1})^*, f_1 \rangle$ and an $(X_{i+1} \times X_{i+2} \times X)$-labeled $(X_i \times X_{i+2})$-tree $\langle (X_i \times X_{i+2})^*, f_2 \rangle$ iff for every $\langle z_1, z'_1 \rangle$ and $\langle z_2, z'_2 \rangle$ in $X_{i-1} \times X_{i+1}$ and for every $\sigma \in (X_{i-1} \times X_{i+1})^*$, we have ($f'$ and f'_1 are the memoryfull versions of f and f'):

- $f(\epsilon) = \langle f_1(\epsilon), f_2(\epsilon) \rangle$.
- $f(\langle z_1, z'_1 \rangle) = \langle f_1(\langle z_0, z'_0 \rangle), f_2(f'_1(\epsilon)) \rangle$.
- $f(\sigma \cdot \langle z_1, z'_1 \rangle \cdot \langle z_1, z'_1 \rangle) = \langle f_1(\langle z_0, z'_0 \rangle \cdot \sigma \cdot \langle z_1, z'_1 \rangle), f_2(f'_1(\langle z_0, z'_0 \rangle \cdot \sigma) \oplus f'(\langle z_0, z'_0 \rangle \cdot \sigma)_{|X_{i+2}}) \rangle$, where $\oplus$ is bitwise concatenation (e.g., $y_1 \cdot y_2 \oplus y_3 \cdot y_4 = \langle y_1, y_3 \rangle \cdot \langle y_2, y_4 \rangle$) and $\tau_{|X_{i+2}}$ is the projection of τ on X_{i+2}.

We then say that $f = f_1 + f_1$. Intuitively, f determines its X_i-element according to f_1 and determines the $(X_{i+1} \times X_{i+2} \times X)$-element by applying

f_2 on an interleaving of an application of f_1', which gives the X_i element and an application of f' on a strict prefix of the input, which returns an element in $X_i \times X_{i+1} \times X_{i+2} \times X$ and is then projected on X_{i+2}. In addition, since we assume that communication involves a delay, f ignores the last letters in a sequence and refers instead to the root directions.

For a set $\mathcal{T}$ of $(X_i \times X_{i+1} \times X_{i+2} \times X)$-labeled $(X_{i-1} \times X_{i+1})$-trees, the set $shape_{X_i \times X_{i+2}}(\mathcal{T})$ consists of all $(X_{i+1} \times X_{i+2} \times X)$-labeled $(X_i \times X_{i+2})$-trees $\langle (X_i \times X_{i+2})^*, f_2 \rangle$ for which there exists an X_i-labeled $(X_{i-1} \times X_{i+1})$-tree $\langle (X_{i-1} \times X_{i+1})^*, f_1 \rangle$ such that $\langle (X_{i-1} \times X_{i+1})^*, f_1 + f_2 \rangle$ is in $\mathcal{T}$.

The shape construction in Theorem 4.1 can be modified to handle the definition of shape above. Essentially, while in the current construction the automaton $\mathcal{A}'$ guesses in each transition a direction x to proceed with, in the new construction $\mathcal{A}'$ needs to guess two elements, corresponding to both X_i and X_{i+2}, and it should remember the X_{i+2} element for the projection described above.

Two-way ring. The solution for two-way rings is based on the modified shape construction described for two-way chains and the "two-direction reasoning" described for one-way rings.

The important common property of the four classes we handle is the fact that there are no two processes both reading input from the envirponmrnt. Consequently, the processes can be linearly ordered according to the signals they know. More architectures fall in this category. For example, it is possible to replace a single processes in a chain by a group of processes that share the same knowladge, and adjust the synthesis algorithms accordingly. An exact characterization of architectures for which the synthesis problem is decidable is an open problem.

6 Discussion

One of the most significant developments in the area of system verification over the last decade is the development of algorithmic methods for verifying temporal specifications of finite-state systems [CGP99]. This derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state systems, as well as from the great ease of use of fully algorithmic methods. A frequent criticism against this approach, however, is that verification is done after significant resources have already been invested in the development of the program. Since systems typically contain errors, verification simply becomes part of the development process. The critics argue that the desired goal is to use the specification in the system development process in order to guarantee the design of correct systems. This is exactly what synthesis algorithms do. Despite this criticism, synthesis tools are not as popular in the industry as verification tools. There are several reasons for that: the scope of synthesis algorithms has been quite limited, their complexity is high, and they do not always produce practical systems, where practicality is measured in a variety of ways, such as optimality (say, number of latches required for implementing the system in hardware, or number of messages needed to be passed between the underlying processes), testability (the ability to test hardware without access to all the internal variables), and the like.

In this paper, we significantly extended the scope of synthesis to include many practical applications. We claim that the high complexity of the problem is not really a serious objection to the potential usefulness of synthesis. First, we note that experience with verification shows that nonelementary algorithms can nevertheless be practical, since the worst-case complexity does not arise often. For example, while the model-checking problem for specifications in second-order logic has nonelementary complexity, the model-checking tool MONA [EKM98, Kla98] successfully verifies many specifications given in second-order logic. Second, we argue that synthesis is not harder than verification. This may sound as a wishful thinking, as it contradicts the known fact that while verification is easy (linear in the size of the model and at most exponential in the size of the specification), synthesis is hard (nonelementary). There is, however, something misleading in this fact: while the complexity of synthesis is given in terms of the specification, the complexity of verification is given with respect to both the specification and the (much bigger) system. In particular, in a distributed setting, it is shown in [Ros92] that there are LTL specifications ψ_n, of length $O(n)$, and architectures with k processes such that the smallest strategy that realizes ψ_n in the given architecture has $k\text{-}exp(n)$ states. What is the complexity of verifying whether a system satisfies ψ_n? Even if verification is linear in the size of the system, it would be nonelementary in n for correct systems, just as the synthesis problem, since such systems necessarily have at least $k\text{-}exp(n)$ states!

In summary, we believe that the real challenge that synthesis algorithms and tools face in the coming years is mostly not that dealing with computational complexity, but rather that of making automatically synthesized systems more practically useful.

References

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th ICALP*, LNCS 372, pp. 1–17, 1989.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Chu63] A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pp. 23–35. institut Mittag-Leffler, 1963.

[Dij72] E.W. Dijksta. *Hierarchical ordering of sequential processes, Operating systems techniques*. Academic Press, 1972.

[Dil89] D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.

[DTV99] M. Daniele, P. Traverso, and M.Y. Vardi. Strong cyclic planning revisited. In S. Biundo and M. Fox, editors, *5th European Conference on Planning*, pp. 34–46, 1999.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th FOCS*, pp. 328–337, 1988.

[EKM98] J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *Proc 10th CAV*, LNCS 1427, pp. 516–520, 1998.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pp. 997–1072, 1990.

[Kla98] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Proc CSL '97*, LNCS, 1997.

[KMTV00] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc. 11th CONCUR*, LNCS 1877, pp. 92–107, 2000.

[KV98] O. Kupferman and M.Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proc. 30th STOC*, pp. 224–233, 1998.

[KV99] O. Kupferman and M.Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245 – 263, June 1999.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Man99] Microsoft LAN Manager. The protocol stack. http://www.rit.edu/~trb5541/p2_stack.html, 1999.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[MS95] D.E. Muller and P.E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141:69–107, 1995.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM TOPLAS*, 6(1):68–93, January 1984.

[PR79] G.L. Peterson and J.H. Reif. Multiple-person alternation. In *Proc. 20st IEEE Symp. on Foundation of Computer Science*, pp. 348–363, 1979.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th POPL*, pp. 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th ICALP*, LNCS 372, pp. 652–671, 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st FOCS*, pp. 746–757, 1990.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical recipes in C*. Cambridge University Press, 1992.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989.

[Tan87] A.S. Tanenboum. *Operating systems, design and implementation*. Prentice-Hall International Editors, New Jersy, 1987.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Proc. 12th TACAS*, LNCS 900, pp. 1–13, 1995.

[Var95] M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. *Proc 7th CAV*, LNCS 939, pp. 267–292, 1995.

Permutation Rewriting and Algorithmic Verification

Ahmed Bouajjani Anca Muscholl Tayssir Touili

LIAFA, Université Paris VII
2, place Jussieu, case 7014
F-75251 Paris Cedex 05
e-mail: {Ahmed.Bouajjani,Anca.Muscholl,Tayssir.Touili}@liafa.jussieu.fr

Abstract

We propose a natural subclass of regular languages (Alphabetic Pattern Constraints, APC) which is effectively closed under permutation rewriting, *i.e., under iterative application of rules of the form* $ab \to ba$. *It is well-known that regular languages do not have this closure property, in general. Our result can be applied for example to* regular model checking, *for verifying properties of parametrized linear networks of regular processes, and for modeling and verifying properties of asynchronous distributed systems.*

We also consider the complexity of testing membership in APC and show that the question is complete for PSPACE when the input is an NFA, and complete for NLOGSPACE when it is a DFA. Moreover, we show that both the inclusion problem and the question of closure under permutation rewriting are PSPACE-complete when we restrict to the class APC.

1 Introduction

Regular languages in their various representations (finite state automata, regular expressions, monadic first or second order logics, temporal logics, etc) are extensively used for modelling and verifying properties of concurrent systems. The main reason is that regular languages enjoy important closure and decidability properties. They were used for modelling behaviors of systems in form of sets of computational sequences, often modulo some abstraction relation [6, 14, 23]. Recently, *regular model checking* was proposed as a technique of symbolic representation of sets of configurations in the analysis of infinite state systems like pushdown automata, fifo-channel systems, and parametrized networks of processes, see e.g. [1, 3, 4, 5, 11, 19, 24]. A fundamental problem which appears in all these areas is then the following one: Given a regular language L and a relation $\mathcal{R}$ on sequences given either by a transducer or a rewriting system, we want to compute – if possible– the set $\mathcal{R}^*(L)$, which is the $\mathcal{R}$-closure of L ($\mathcal{R}^*$ denotes the reflexive, transitive closure of $\mathcal{R}$). Since unrestricted rewriting systems have full computational power, we have to impose restrictions on the rewriting rules and on the regular languages we consider, in order to be able to compute $\mathcal{R}^*(L)$. In this paper we consider permutation rewriting rules of the form $ab \to ba$, where a, b are letters of a given alphabet Σ. Such rewriting rules are usually called *semi-commutation rules* in Mazurkiewicz trace theory [7]. Our primary goal is to determine a suitable subclass of regular languages for which we can effectively compute the $\mathcal{R}$-closure, for *any* semi-commutation rewriting system $\mathcal{R}$.

The problem of computing the closure of a language under a semi-commutation rewriting systems appears naturally in several areas. For instance, partial-order reduction methods [9, 17, 22] applied in traditional model-checking rely on the fact that the property we want to verify does not distinguish different linearizations of the same partial order. This allows to perform an improved, reduced exploration of large systems. In the simplest setting, a partial-order property means that the property is closed under *partial commutation rules*, i.e., (symmetric) rules of the form $ab \leftrightarrow ba$, meaning that two actions a and b are causally independent. However, it is often much more convenient to express a prop-

0-7695-1281-X/01 $10.00 © 2001 IEEE

erty (or its negation) as a set of behaviors (or bad behaviors), regardless of all possible interleavings between independent actions. Therefore, if a given property ϕ is not a partial-order property, then we can first compute its closure $\mathcal{R}^*(\phi)$. The interest in doing this is that closing ϕ is in general much less expensive than a full exploration of the system.

In the context of *regular model checking* [5, 11, 19], a set of configurations is represented as a regular language and the actions of a system are modeled as a rewriting system $\mathcal{R}$. Then, the verification problem amounts to compute the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ for a given set of initial configurations L. This allows for instance to analyze parameterized systems with arbitrarily many identical finite state processes which are connected linearly. Here, a configuration is a sequence of control states of individual processes, the i-th element of the sequence being the state of the i-th process. Thus, sets of configurations of arbitrary lengths, corresponding to systems with arbitrary number of processes, are described by a regular language. This allows a *uniform verification*, i.e., for any number of processes. In protocols based on information exchange between neighbors (e.g., token exchange, mutual exclusion, leader election), certain transitions can be modeled by semi-commutation rewriting rules of the form $ab \rightarrow ba$. Being able to compute the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ allows for instance to compute the effect of meta-transitions corresponding to the semi-commutation rewriting rules. Take as an example a simple mutual exclusion protocol, where linearly ordered processes can exchange a token which gives the right to enter a critical section. Suppose that the state of a process is 1 if it owns the token, and 0 otherwise. The initial configuration is then the regular expression 10^* (note that the number of processes is not fixed). An (abstract) transition rule of the system can be represented by the semi-commutation one-rule system $\mathcal{R} = \{10 \rightarrow 01\}$. We can now compute the reachable set of configurations $\mathcal{R}^*(10^*) = 0^*10^*$ and check for instance that the intersection with the set of bad configurations $(0+1)^*1(0+1)^*1(0+1)^*$ is empty.

Thus, given a regular language L and a *semi-commutation relation* $\mathcal{R}$, we want to compute the reflexive, transitive closure $\mathcal{R}^*(L)$. However, it is not hard to see that semi-commutation rewriting does not preserve regularity. In our setting we would like to have a subclass of regular languages which is effectively closed under several operations, such as *union*, *intersection* and *semi-commutation rewriting*. Closure under these operations allows us to perform automatically a sequence of operations as required for example in the iterative fixed point computations of regular model checking. Clearly, we want a subclass of regular languages with a decidable membership problem. The solution proposed by this paper is the class of *Alphabetic Pattern Constraints* (APC), which appears naturally in many contexts of verification of concurrent systems. APC corresponds to finite unions of languages of the form $\Sigma_0^* a_1 \Sigma_1^* \cdots a_n \Sigma_n^*$, where every Σ_i denotes a subset of the alphabet Σ and every $a_i \in \Sigma$ denotes a single letter. For instance, the regular expressions in the token ring example above are APC expressions. APCs can be used for example for (negated) safety properties expressing the presence of patterns within computations or configurations, such as required for mutual exclusion. The class of APCs actually corresponds to the Σ_2-level of the quantifier-alternation hierarchy of the first-order logic of sequences [21]. We show that this class satisfies all the closure properties stated above. In particular, our first main result is that APC is closed under semi-commutation rewriting and we provide an effective algorithm that computes the closure $\mathcal{R}^*(L)$, given a semi-commutation system $\mathcal{R}$ and an APC language L.

For regular model checking we consider also circular semi-commutation rewriting. Indeed, the simplest interconnection topology in distributed computing is the ring topology. A (parameterized) configuration corresponds then to a circular word, i.e. a word $x_1 \cdots x_n$ with the understanding that x_1 follows x_n. This means that $x_1 \cdots x_n$ and its *conjugated* words $x_k x_{k+1} \cdots x_n x_1 \cdots x_{k-1}$ represent the same configuration. Thus, the set of configurations of a ring network is a set of words L which is closed under conjugacy, i.e. $L = \mathrm{Conj}(L)$. For instance, for the Token Ring Protocol the set of initial configurations on a ring is $\mathrm{Conj}(10^*) = 0^*10^*$. Our second main result shows that for any semi-commutation rewriting system $\mathcal{R}$, the circular $\mathcal{R}$-closure $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L)$ of any language $L \subseteq \Sigma^*$ can be computed as long as the reflexive, transitive closure $\mathcal{R}^*(L)$ is computable. For this we show that $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L) = (\mathrm{Conj} \circ \mathcal{R}^*)^{2^{|\Sigma|}}(L)$. This implies that for each APC language L the circular $\mathcal{R}$-closure $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L)$ is in APC and can be effectively computed.

In the last part of this paper we establish complexity bounds for basic problems concerning the

class of APC languages. We show that deciding whether a regular language belongs to APC is complete for PSPACE when the language is given by a non-deterministic automaton, respectively complete for NLOGSPACE, when the input is a deterministic automaton. Moreover, we show that testing whether an APC language is closed under a semi-commutation rewriting relation, as well as the inclusion problem for APC, are both PSPACE-complete problems. These results suggest that APC is as "hard" as the whole class of regular languages, which means in some sense that APCs are expressive enough for specifying interesting properties. It is also interesting to note that APCs correspond to the smallest level in the quantifier-alternation hierarchy of first-order logic which has this "hardness property". Indeed, languages in Σ_1 and Π_1 correspond respectively to upward and downward subword-closed sets. For example, Π_1 is precisely the class SRE [1], for which it can be shown that inclusion can be checked in polynomial time.

Related work: Problems related to closure of languages under semi-commutations have been studied in the community of trace theory (see e.g. chapter 12 in [7] for a survey). However, the problems addressed here and our results have a different flavor. Our aim is to identify subclasses of regular languages which are closed under *all* semi-commutation rewriting relations, whereas classical results of trace theory aim at providing for a given semi-commutation relation $\mathcal{R}$ sufficient conditions on regular languages L ensuring that the $\mathcal{R}$-closure of L remains regular. Moreover, these conditions on the languages always depend on the relation $\mathcal{R}$.

APC languages have been intensively studied in logic and algebra. As mentioned above, they correspond to the Σ_2-level of the quantifier-alternation hierarchy of first order logic, i.e., to formulas of the form $\exists^*\forall^*\phi$, where ϕ is quantifier-free. The class APC has also an algebraic characterization, it corresponds to level 3/2 of Straubing's concatenation hierarchy of star-free sets. Moreover, it is the largest hierarchy level known to be decidable [18].

The complexity of deciding whether a regular ω-language is closed under commutation rewriting was considered in [16, 20]. Several works on regular model checking deal with the problem of computing the closure of a regular language under a rewriting system [2, 5, 8, 10, 19]. However, the techniques proposed in these papers are not complete, in general. Moreover, they do not cover the case of semi-commutation rewriting.

2 Alphabetic Pattern Constraints

In this section we define the class of *Alphabetic Pattern Constraints (APC)* and show that APC is closed under union, intersection and conjugacy, but not under complementation.

Definition 2.1 *Let Σ be a finite alphabet. An atomic expression over Σ is either a letter a of Σ or a star expression $(a_1+a_2+\cdots+a_n)^*$, where $a_1, a_2, \ldots, a_n \in \Sigma$. The set of star expressions is denoted by $S(\Sigma)$.*

A product p over Σ^* is a (possibly empty) concatenation $e_1e_2\cdots e_n$ of atomic expressions $e_1, \ldots, e_n$ over Σ. We use ϵ to denote the empty product.

An Alphabetic Pattern Constraint (APC) over Σ^* is an expression of the form $p_1+\cdots+p_n$, where $p_1, \ldots, p_n$ are products over Σ^*. By APC(Σ) we denote the set of regular languages described by some APC over Σ^*.

In the rest of the paper we will not distinguish between a regular expression and the language that it defines. However, the input for our algorithms in Sections 3, 5 will be an APC expression.

It can be easily noted that the class of APCs is not closed under complementation. Consider for example the alphabet $\Sigma = \{a, b\}$ and the APC language $\Sigma^*aa\Sigma^* + \Sigma^*bb\Sigma^* + b\Sigma^* + \Sigma^*a$. It is not difficult to check that its complement $(ab)^*$ does not belong to APC.

Let us introduce some notations which will be used in the analysis of operations on APCs. Let $p = e_1 \cdots e_n$ be a product, then the length of p, denoted $l(p) = n$, is the number of atomic expressions in p. Let $e = \sum_i p_i$ be an APC expression, then the *length* of e is defined as $l(e) = \max_i l(p_i)$. The *size* of an expression is the sum of the lengths of its products. For a language L we denote by $\alpha(L)$ the set of letters of Σ appearing in L. As usual, $|L|$ denotes the cardinality of L. For a string $w \in \Sigma^*$ and a letter $a \in \Sigma$, we denote by $|w|_a$ the number of occurrences of a in w.

We recall that two words x and $y \in \Sigma^*$ are called *conjugated* if $x = uv$ and $y = vu$ for some $u, v \in \Sigma^*$. For a language L, we denote by $\mathrm{Conj}(L)$ the set $\{uv \in \Sigma^* \mid vu \in L\}$ of conjugates of words

from L. For a class of languages $\mathcal{C}$ to be *closed under conjugacy* we require that $L \in \mathcal{C}$ implies that $\mathrm{Conj}(L) \in \mathcal{C}$.

We conclude this section by stating some straightforward closure properties of APC. The proofs are not difficult and can be found in the full version of the paper.

Proposition 2.1 *The class APC is closed under union, intersection and conjugacy.*

Remark 2.1 *While union and conjugacy are polynomial operations, computing the intersection of two APC languages yields an expression of exponential size. The worst-case is indeed exponential, as shown by the following example. Consider the products $p_n = b^*(ab^*)^n$ and $q_n = (a^*b)^n a^*$ each of size $2n+1$. Then $\{w \in (a+b)^* \mid |w|_a = |w|_b = n\} = p_n \cap q_n$ is a finite set with the property that every APC expression for $p_n \cap q_n$ is of exponential size.*

3 Semi-Commutation Rewriting and APC

Semi-commutations are a natural way of expressing causal independence in concurrent systems in an algebraic way. The original notion was proposed in the late 70's by Mazurkiewicz [12] for the semantics of Petri nets. Mazurkiewicz traces and semi-traces are a model of true concurrency with nice algorithmical properties, which can be exploited for automatic verification methods.

A semi-commutation relation $\mathcal{R}$ defined over an alphabet Σ of actions is an irreflexive binary relation, i.e., a subset of $\Sigma \times \Sigma \setminus \{(a,a) \mid a \in \Sigma\}$. The idea is that two actions a, b with $(a,b) \in \mathcal{R}$ are (partially) causally independent, in the sense that we can rewrite ab into ba in every context. In many cases the relation $\mathcal{R}$ is asymmetric, for instance in a producer-consumer model we may rewrite $cp \to pc$, but not the other way round.

It is not difficult to see that semi-commutation rewriting does not preserve regulairty. Consider for example the set $L = (ab)^*$ and the semi-commutation system $\mathcal{R} = \{ba \to ab\}$. Then, $\mathcal{R}^*(L)$ is the (non-regular) set of all words having the same number of a's and b's, and such that all their prefixes contain at least as many a's as b's. Therefore, we cannot hope to represent the relation $\mathcal{R}^*$ by a finite transducer, in general.

We associate with each semi-commutation relation $\mathcal{R}$ a rewriting relation $\rho_{\mathcal{R}} \subseteq \Sigma^* \times \Sigma^*$, which is defined by $(w, w') \in \rho_{\mathcal{R}}$ if there exist $w_1, w_2 \in \Sigma^*$ and $a, b \in \Sigma$ such that $(a,b) \in \mathcal{R}$, $w = w_1 abw_2$, and $w' = w_1 baw_2$. As usual, we denote by $\rho^*_{\mathcal{R}}$ the reflexive, transitive closure of $\rho_{\mathcal{R}}$. For a language $L \subseteq \Sigma^*$, we denote its $\mathcal{R}$-*closure* $\{v \in \Sigma^* \mid \exists u \in L, (u,v) \in \rho^*_{\mathcal{R}}\}$ by $\mathcal{R}^*(L)$.

The notation of semi-commutations can be extended to sets by letting for each subsets $X, Y \subseteq \Sigma$:

$$(X,Y) \in \mathcal{R} \text{ if } X \times Y \subseteq \mathcal{R}\,.$$

Let $\mathcal{R}$ be a semi-commutation relation, then we denote by $\delta_{\mathcal{R}}$ the value

$$\delta_{\mathcal{R}} = \max_{a \in \Sigma}\{|Y| \mid Y \subseteq \Sigma \text{ such that } (a,Y) \in \mathcal{R}\}\,.$$

We will assume throughout the paper that $\mathcal{R} \neq \emptyset$, thus $\delta_{\mathcal{R}} > 0$.

Our first main result is stated in the theorem below. The remaining of this section consists in describing the algorithm underlying Theorem 3.1. Several proofs are omitted and can be found in the full version of the paper.

Theorem 3.1 *For each APC expression L, the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ belongs to APC and can be computed effectively. Moreover, the length of the computed expression is in $\mathcal{O}((\delta_{\mathcal{R}}+1)^{l(L)})$.*

Since $L \in \mathrm{APC}(\Sigma)$ is a finite union of products, its closure $\mathcal{R}^*(L)$ is the union of closures of its products. Hence, it suffices to show how to compute effectively $\mathcal{R}^*(p)$ for a given product p. For this we use the $\mathcal{R}$−shuffle operation defined below. The idea is to compute $\mathcal{R}^*(e_1 \cdots e_n)$ recursively, i.e., computing first $\mathcal{R}^*(e_2 \cdots e_n)$ and using that $\mathcal{R}^*(e_1) = e_1$. The recursive step means that we need to compute $\mathcal{R}^*(eL)$, for an $\mathcal{R}$-closed APC expression L and an atomic expression e, an operation which will be performed also recursively. For our computations we need the following notations:

Definition 3.1 *Let $\mathcal{R}$ be a semi-commutation relation. Given two words x and y of Σ^*, the $\mathcal{R}$−shuffle of x and y, denoted by $x \amalg_{\mathcal{R}} y$, is the set of words of the form $x_1 y_1 \cdots x_n y_n$ with $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, $x_i, y_i \in \Sigma^*$ for all $1 \leq i \leq n$ and such that $(\alpha(x_i), \alpha(y_j)) \in \mathcal{R}$ for all $j < i$.*

The $\mathcal{R}$−shuffle extends to sets $X, Y \subseteq \Sigma^*$ by letting

$$X \amalg_{\mathcal{R}} Y = \{x \amalg_{\mathcal{R}} y \mid x \in X, y \in Y\}.$$

Note that for all $x, y \in \Sigma^*$, we have $\mathcal{R}^*(xy) = \mathcal{R}^*(x) \amalg_{\mathcal{R}} \mathcal{R}^*(y)$. The next lemma shows how to compute $\mathcal{R}^*(LK)$ when L and K are already $\mathcal{R}$-closed.

Lemma 3.1 *Let L and K be two $\mathcal{R}$-closed sets, i.e., we suppose that we have both $\mathcal{R}^*(L) = L$ and $\mathcal{R}^*(K) = K$. Then we have $\mathcal{R}^*(LK) = L \amalg_{\mathcal{R}} K$.*

Since any atomic expression is $\mathcal{R}$-closed we can state the following lemma:

Lemma 3.2 *Let $e_1, e_2, \ldots, e_n$ be atomic expressions and let $p = e_1 e_2 \cdots e_n$ be a product, then we have:*

$$\mathcal{R}^*(p) = e_1 \amalg_{\mathcal{R}} (e_2 \amalg_{\mathcal{R}} (\cdots (e_{n-1} \amalg_{\mathcal{R}} e_n) \cdots)) .$$

By the preceding lemma we can compute $\mathcal{R}^*(p)$ recursively. Lemma 3.3 and Proposition 3.1 below are the basic cases of our algorithm.

Lemma 3.3 *Let E be a subset of Σ and let $a \in \Sigma$ be a letter, then we have:*

$$E^* \amalg_{\mathcal{R}} a = \mathcal{R}^*(E^* a) = E^* a E'^* ,$$

where $E' = \{x \in E \mid (x, a) \in \mathcal{R}\}$.

Example 3.1 *Consider the product $p = (e + f + g)^* d$, and the semi-commutation relation $\mathcal{R}_1 = \{(f, d), (g, d)\}$. Then the previous lemma yields*

$$\mathcal{R}_1^*(p) = (e + f + g)^* \amalg d = (e + f + g)^* d (f + g)^* .$$

The next proposition is the main technical result needed for the proof of Theorem 3.1. It shows that the $\mathcal{R}$-closure of the product of two star expressions belongs to APC. In particular, note that the length of the products in the expression given below is bounded above by a constant which is polynomial in Σ and $\mathcal{R}$.

Proposition 3.1 *Let E and F be two subsets of Σ, then $E^* \amalg_{\mathcal{R}} F^* = \mathcal{R}^*(E^* F^*)$ equals*

$$\sum E^*(E_1 + F_1)^* \cdots (E_n + F_n)^* F^* ,$$

where the sum is taken over all subsets E_i and F_i of Σ satisfying the following conditions:

- $\emptyset \neq E_n \subsetneq \cdots \subsetneq E_1 \subseteq E$,
- $\emptyset \neq F_1 \subsetneq \cdots \subsetneq F_1 \subseteq F$,
- $(E_i, F_j) \in \mathcal{R}$ *for all* $1 \leq j \leq i \leq n$.

Proof. The first equality can be inferred as previously from Lemma 3.1 since E^* and F^* are closed under $\mathcal{R}$.

Let us consider now the second equality. It is obvious that $E^*(E_1+F_1)^* \cdots (E_n+F_n)^* F^* \subseteq \mathcal{R}^*(E^* F^*)$, whenever E_i and F_i satisfy $(E_i, F_j) \in \mathcal{R}$ for all $j \leq i$.

Conversely, let $w \in E^* \amalg_{\mathcal{R}} F^* = \mathcal{R}^*(E^* F^*)$. We can write $w = u_1 v_1 u_2 v_2 \cdots u_m v_m$ with $u_i \in E^*, v_i \in F^*$, and such that $(\alpha(u_i), \alpha(v_j)) \in \mathcal{R}$ holds for all $j < i$. Clearly, we can assume that $u_i, v_j \neq \epsilon$ for all $i \neq 1$ and $j \neq m$.

Consider the sequences $(k_i)_{1 \leq i \leq n}$, $(E_i)_{1 \leq i \leq n}$ and $(F_i)_{1 \leq i \leq n}$ defined inductively by:

- $k_1 = 1$, $k_i = \min\{j \mid k_{i-1} < j < m, v_j \notin F_{i-1}^*\}$,
- $E_i = \alpha(u_{k_i+1} \cdots u_m)$,
- $F_i = \{y \in F \mid \forall x \in E_i, (x, y) \in \mathcal{R}\}$.

By definition we have $E_{i+1} \subsetneq E_i \subseteq E$, hence $F_i \subsetneq F_{i+1} \subseteq F$ for all i. Moreover, $(E_i, F_i) \in \mathcal{R}$ holds for all i, therefore $(E_i, F_j) \in \mathcal{R}$ for all $j \leq i$. Finally, we note that $u_{k_i+1} \cdots u_{k_{i+1}} \in E_i^*$ and $v_{k_i} \cdots v_{k_{i+1}-1} \in F_i^*$, which yields the result. □

Remark 3.1 *Note that the cardinality of E_1 is at most $\delta_{\mathcal{R}}$, since we require that $F_1 \neq \emptyset$ and $(E_1, F_1) \in \mathcal{R}$. Moreover, since there is a strict inclusion between the E_i's, the length of the products in the expression for $\mathcal{R}^*(E^* F^*)$ is at most $\delta_{\mathcal{R}} + 2$.*

Example 3.2 *Consider the product $p = (a + b + c)^*(e + f + g)^*$, and the semi-commutation relation $\mathcal{R}_2 = \{(a, e), (c, g), (b, e), (b, f)\}$. From the proposition above it follows that $\mathcal{R}_2^*(p) = (a + b + c)^* \amalg_{\mathcal{R}_2} (e + f + g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* + (a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^*$.*

We are now going to compute effectively $\mathcal{R}^*(p) = \mathcal{R}^*(e_1 e_2 \cdots e_n)$ and show that it belongs to APC. By Lemma 3.3 and Proposition 3.1 we have shown the result for $n = 2$. Suppose now that $\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 f_2 \cdots f_k$, with f_i denoting atomic expressions, and let us show that $\mathcal{R}^*(e_1 e_2 \cdots e_n)$, which equals $\sum e_1 \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_k)$, also belongs to APC. Thus, we only need to compute $e_1 \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ and to show that it is of the required form. To do this we will distinguish two cases, depending on e_1

being a letter or a star expression. The first case is straightforward:

Lemma 3.4 *Let $a \in \Sigma$ and $f_1, \ldots, f_n$ be atomic expressions, then*

$$a \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_n) = \sum_j g_1 \cdots g_j a h_j f_{j+1} \cdots f_n$$

such that, for all $i \leq j$ we have:

- *if $f_i \in S(\Sigma)$, then $g_i \in S(\Sigma)$ with $\alpha(g_i) = \{x \in \alpha(f_i) \mid (a, x) \in \mathcal{R}\}$,*
- *if $f_i = b \in \Sigma$ and $(a, b) \in \mathcal{R}$, then $g_i = b$.*

Moreover, $h_j = f_j$ when $f_j \in S(\Sigma)$ and $h_j = \varepsilon$ when $f_j \in \Sigma$.

Example 3.3 *Let $\mathcal{R}_3$ be the semi-commutation relation $\mathcal{R}_3 = \{(h, a), (h, e)\}$. Then the previous lemma implies that $h \amalg_{\mathcal{R}_3} (a+b+c)^*(a+b+c)^*(b+e+f)^* = a^* h (a+b+c)^*(a+b+e)^*(b+e+f)^* + (a+e)^* h (a+b+e)^*(b+e+f)^*$.*

The next proposition generalizes Lemma 3.3 and Proposition 3.1.

Proposition 3.2 *Let E and F be two subsets of Σ, $a \in \Sigma$ a letter, and L be a language of Σ^*, then we have:*

1. *$E^* \amalg_{\mathcal{R}} (aL) = (E^* \amalg_{\mathcal{R}} a)(E'^* \amalg_{\mathcal{R}} L)$, where $E' = \{b \in E \mid (b, a) \in \mathcal{R}\}$.*
2. *$E^* \amalg_{\mathcal{R}} (F^* L)$ equals*

$$\sum_{\substack{(E', F') \in \mathcal{R} \\ E' \subseteq E, F' \subseteq F}} (E^* \amalg_{\mathcal{R}} F'^*)(E'^* \amalg_{\mathcal{R}} L).$$

Corollary 3.1 *Let E and F be two subsets of Σ, and let L be a language of Σ^*, then $E^* \amalg_{\mathcal{R}} (F^* L)$ equals:*

$$\sum E^*(E_1 + F_1)^*(E_2 + F_2)^* \cdots (E_k + F_k)^*(E_k^* \amalg_{\mathcal{R}} L),$$

where the union is taken over all subsets E_i and F_i of Σ satisfying:

- $E_k \subsetneq \cdots \subsetneq E_1 \subseteq E$,
- $\emptyset \neq F_1 \subsetneq \cdots \subsetneq F_k \subseteq F$,
- $(E_i, F_j) \in \mathcal{R}$ *for all* $1 \leq j \leq i \leq k$.

Proof. The inclusion from right to left is straightforward. By Proposition 3.2 it remains to show that

$$(E^* \amalg_{\mathcal{R}} F'^*)(E'^* \amalg_{\mathcal{R}} L) \subseteq \sum E^*(E_1 + F_1)^* \cdots (E_k + F_k)^*(E_k^* \amalg_{\mathcal{R}} L),$$

where $E' \subseteq E$ and $F' \subseteq F$ are subsets satisfying $(E', F') \in \mathcal{R}$. This can be obtained from Proposition 3.1 applied to $E^* \amalg_{\mathcal{R}} F'^*$, by noting that the sequence of $(E_i)_i$ can be chosen such that each E_i is maximal with the property that $(a, b) \in \mathcal{R}$ for all $a \in E_i, b \in F_i$. Hence, $E' \subseteq E_i$ for all i yields the claimed expression.

□

Example 3.4 *Let $\mathcal{R}_4$ be the semi-commutation relation $\mathcal{R}_4 = \{(a, e), (c, g), (b, e), (b, f), (a, d)\}$. Then from the last proposition and from example 3.2 it follows that*

$$(a+b+c)^* \amalg_{\mathcal{R}_4} (e+f+g)^* d (f+g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* d (f+g)^* + (a+b+e)^*(a+b+e)^*(b+e+f)^*(e+f+g)^* d (f+g)^* + (a+b+c)^*(a+b+e)^* da^*(f+g)^*.$$

Proposition 3.2 and Corollary 3.1 yield the recursive step for computing $E^* \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_n)$:

Proposition 3.3 *Let $E \subseteq \Sigma$ and let $f_1, \ldots, f_n$ be atomic expressions. Then $E^* \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ equals*

1. *For a star expression f_1: $\sum E^*(E_1 + F_1)^* \cdots (E_k + F_k)^*(E_k^* \amalg_{\mathcal{R}} f_2 \cdots f_n)$, where the union is taken over all subsets E_i, F_i satisfying $E_{i+1} \subsetneq E_i \subseteq E$, $\emptyset \neq F_i \subsetneq F_{i+1} \subseteq \alpha(f_1)$ and $(E_i, F_j) \in \mathcal{R}$ for all $j \leq i$.*
2. *For a single letter $f_1 = a$:*

$$E^* a (E'^* \amalg_{\mathcal{R}} f_2 \cdots f_n).$$

We can now describe the algorithm for computing the closure of an APC expression $\sum e_1 \cdots e_n$ under a semi-commutation rewriting relation $\mathcal{R}$. We compute recursively $\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 \cdots f_k$. The recursive step is given by Lemma 3.4, if e_1 is a letter. Otherwise, for $e_1 = E^*$ we apply Proposition 3.3, which is itself a recursive algorithm. It is easily seen that each step preserves containment in APC. This shows Theorem 3.1.

Remark 3.2 *We note that for a product p of length n the length of the products of the expression computed for $\mathcal{R}^*(p)$ is at most $\mathcal{O}((\delta_{\mathcal{R}}+1)^n)$. Moreover, since there exist $2^{|\Sigma|}+|\Sigma|$ different atomic expressions, it follows that the size of $\mathcal{R}^*(p)$ is at most $2^{O(|\Sigma|(\delta_{\mathcal{R}}+1)^n)}$.*

4 Applications

As mentioned in the introduction, we can use our results for applying partial-order reduction techniques in model-checking, even if the original property is not a partial-order property. This idea can be used for example in the validation of scenarios described by High-Level Message Sequence Charts (HMSC). HMSCs are a graphical specification language for communications protocols, standardized by the ITU and integrated in UML. An HMSC scenario is a partial-order model for asynchronous fifo message exchange of concurrent processes. Assume for example that we have a system S including two processes P and Q and that we want to verify that P cannot send more than two messages to Q before getting an acknowledgement back from Q. Let us denote the set of possible actions by Σ, the send action of P to Q by s, the receive action of Q from P by r and let Σ_P (resp. Σ_Q) denote events on P (resp. on Q). Hence, a bad scenario contains for example an occurrence of the sequence $srsr$, which means that two messages have been transmitted from P to Q without an acknowledgement between them. So, let $\phi = \Sigma^* srsr \Sigma^*$ be the set of sequences containing this bad subsequence, and suppose we want to verify that an HMSC system S satisfies $\neg\phi$. Clearly, ϕ is not a partial-order property, since ϕ does not contain for example the sequence $ssrr$. We can consider the semi-commutation rule $rs \to sr$ which expresses that communication is asynchronous. By applying our algorithm with suitable rules such as $\mathcal{R} = \{rs \to sr\}$ we obtain the partial-order property $\mathcal{R}^*(\phi)$:

$$\Sigma^* s(\Sigma \setminus \Sigma_P)^* r(\Sigma \setminus (\Sigma_P \cup \Sigma_Q))^* s(\Sigma \setminus \Sigma_Q)^* r\Sigma^* + \Sigma^* s(\Sigma \setminus \Sigma_P)^* s\Sigma^* r(\Sigma \setminus \Sigma_Q)^* r\Sigma^* .$$

Now, for verifying that S satisfies $\neg\phi$ we can consider a succinct representation of the system S, which corresponds to the transition system underlying S and which is polynomial in the size of the given HMSC system, and then check that $S \cap \mathcal{R}^*(\phi)$ is empty. Since we consider an $\mathcal{R}$-closed property, it is not necessary to compute the closure of the system S, which is an expensive operation, and even impossible in general (linearizations of HMSCs are not regular [13]). The same holds also in the case of "positive reasoning": for verifying that S satisfies a property ϕ, it suffices to construct the $\mathcal{R}$-closure of ϕ and check that $S \subseteq \mathcal{R}^*(\phi)$.

Further examples showing that APC properties occur naturally in the verification of concurrent systems is the so-called "matching with gaps" problem in HMSCs [15], which is a kind of weak model-checking. Other examples from distributed computing are negations of (some) safety properties when APCs are used to express bad patterns (scenarios) like in the examples shown above. Furthermore, in the context of regular model checking, it turns out that the reachability sets of many infinite-state systems and parameterized systems, including communication protocols like the alternating bit and the sliding window, and parameterized mutual exclusion protocols such as the token ring, Szymanski's, Burns', or Dijkstra's protocols, are all expressible as APCs. Being able to compute the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ for a semi-commutation system $\mathcal{R}$ allows us to compute the effect of meta-transitions corresponding to the semi-commutation rules, and hence to accelerate the process of computing the set of reachable configurations.

5 Circular Rewriting

In this section we consider the problem of computing $\mathcal{R}^*(L)$ when L consists of circular words. This amounts to assume that L is closed under conjugacy, $L = \mathrm{Conj}(L)$. Recall that $\mathrm{Conj}(L) = \{vu \mid uv \in L\}$ denotes the closure of L under conjugacy. The question of computing the $\mathcal{R}$-closure in this framework arises naturally in regular model checking when processes are ordered circularly in a ring.

Let $\mathcal{R} \subseteq \Sigma \times \Sigma$ be a semi-commutation relation over Σ. We associate with $\mathcal{R}$ the *circular rewriting relation* $\mathcal{R}_c \subseteq \Sigma^* \times \Sigma^*$ defined as follows. For any pair of words x and y in Σ^*, we define $(x, y) \in \mathcal{R}_c$ if we can write

$$uv \in \mathcal{R}^*(x) \text{ and } y \in \mathcal{R}^*(vu),$$

for some $u, v \in \Sigma^*$. Note that the circular rewriting relation $\mathcal{R}_c$ is the composition of the (rewriting) relations $\mathcal{R}^* \circ \mathrm{Conj} \circ \mathcal{R}^*$. As usual, $\mathcal{R}_c^*$ denotes the reflexive, transitive closure of $\mathcal{R}_c$. For a language

L we denote by $\mathcal{R}_c^*(L)$ the *circular $\mathcal{R}$-closure* of L, defined as the set:

$$\mathcal{R}_c^*(L) = \{v \in \Sigma^* \mid \exists\, u \in L \text{ such that } (u,v) \in \mathcal{R}_c^*\}.$$

We will show in this section that the circular $\mathcal{R}$-closure $\mathcal{R}_c^*(L)$ of *any* language L (not necessarily regular) can be obtained by applying alternatively conjugation and permutation rewriting a finite number of times.

The main result of this section can be stated as follows:

Theorem 5.1 *Let $L \subseteq \Sigma^*$, then $\mathcal{R}_c^*(L) = \mathcal{R}_c^{2|\Sigma|}(L)$.*

As a first corollary, we obtain the closure of the class APC under circular rewriting.

Corollary 5.1 *Let L be a APC expression, then $\mathcal{R}_c^*(L)$ is in APC and is effectively computable. The length of the expression computed for $\mathcal{R}_c^*(L)$ is at most $(\delta_{\mathcal{R}}+1)^{\mathcal{O}(l(L)|\Sigma|)}$.*

Proof. This follows directly from $\mathcal{R}_c^*(L) = \mathcal{R}_c^{2|\Sigma|}(L) = (\mathcal{R}^* \circ \mathrm{Conj} \circ \mathcal{R}^*)^{2|\Sigma|}(L)$, together with and APC($\Sigma$) being closed under semi-commutation rewriting and conjugacy (Theorem 3.1 and Proposition 2.1). □

In the remaining of the section we show Theorem 5.1. The proof uses ideas from [7][Ch. 3]. It generalizes (and simplifies) the proof given there for the case where $\mathcal{R}$ is a symmetric relation. As in [7] we need a second relation $\mathcal{C}_{\mathcal{R}}$, called *conjugacy relation*, which is defined as follows for $x, y \in \Sigma^*$:

$$(x,y) \in \mathcal{C}_{\mathcal{R}} \text{ if } \exists\, z \in \Sigma^* \text{ such that } zy \in \mathcal{R}^*(xz)\,.$$

Lemma 5.1 *$\mathcal{R}_c \subseteq \mathcal{C}_{\mathcal{R}}$ and $\mathcal{C}_{\mathcal{R}}$ is reflexive and transitive.*

Proof. For the first claim let $x, y \in \Sigma^*$ be such that $(x,y) \in \mathcal{R}_c$. By definition, there exist u and $v \in \Sigma^*$ such that $uv \in \mathcal{R}^*(x)$ and $y \in \mathcal{R}^*(vu)$, then $uy \in \mathcal{R}^*(uvu) \subseteq \mathcal{R}^*(xu)$.

For the second claim it is easy to see that $\mathcal{C}_{\mathcal{R}}$ is reflexive. Let now $x, y, z \in \Sigma^*$ be such that $(x,y) \in \mathcal{C}_{\mathcal{R}}$ and $(y,z) \in \mathcal{C}_{\mathcal{R}}$. Let then w and $t \in \Sigma^*$ be such that $wy \in \mathcal{R}^*(xw)$ and $tz \in \mathcal{R}^*(yt)$. Then, $(wt)z \in \mathcal{R}^*(wyt) \subseteq \mathcal{R}^*(x(wt))$, which shows that $(x,z) \in \mathcal{C}_{\mathcal{R}}$. □

Theorem 5.2 *Let $x, y \in \Sigma^*$. Suppose that $z \in \Sigma^*$ is such that $zy \in \mathcal{R}^*(xz)$. Then there exist $m \leq 2|\Sigma|$, and words $t_0, \ldots, t_m \in \Sigma^*$ satisfying the following properties:*

- $t_0 \cdots t_m \in \mathcal{R}^*(x)$,
- $y \in \mathcal{R}^*(t_m \cdots t_0)$,
- $(\alpha(t_j), \alpha(t_i)) \in \mathcal{R}$ *for all* $j > i+1$.

Proof. We only sketch the proof idea. We suppose that $zy \in \mathcal{R}^*(xz)$. Then a combinatorial lemma (Levi's Lemma for semi-traces, see [7][Ch. 12]) implies that there exist words $u, v, p, q \in \Sigma^*$ such that $up \in \mathcal{R}^*(x)$, $qv \in \mathcal{R}^*(z)$, $z \in \mathcal{R}^*(uq)$, and $y \in \mathcal{R}^*(pv)$, such that $(\alpha(p), \alpha(q)) \in \mathcal{R}$. Since $qv \in \mathcal{R}^*(uq)$ and $|u|+|q| < |x|+|z|$ if x is nonempty, we can apply induction on $|x|+|z|$ in order to obtain the result. □

Corollary 5.2 $\mathcal{R}_c^* = \mathcal{R}_c^{2|\Sigma|}$.

Proof. First, we show that $\mathcal{C}_{\mathcal{R}} \subseteq \mathcal{R}_c^{2|\Sigma|}$. Let $(x,y) \in \mathcal{C}_{\mathcal{R}}$ with $zy \in \mathcal{R}^*(xz)$ for some z. Let $t_0, \ldots, t_p$ be as stated in the Theorem 5.2. It suffices to show that $(t_0 \cdots t_p, t_p \cdots t_0) \in \mathcal{R}_c^{2|\Sigma|}$. This is due to the fact that $(t_0 t_p \cdots t_1, t_p \cdots t_0) \in \mathcal{R}_c$ and that for each $i \in \{1, \ldots, p-1\}$:

$$(t_0 \cdots t_i t_p \cdots t_{i+1}, t_0 \cdots t_{i-1} t_p \cdots t_i) \in \mathcal{R}_c$$

since

$$t_0 \cdots t_{i-1} t_p \cdots t_i \in \mathcal{R}^*(t_p \cdots t_{i+1} t_0 \cdots t_i).$$

Indeed, to obtain the word $t_0 \cdots t_{i-1} t_p \cdots t_i$ from $t_p \cdots t_{i+1} t_0 \cdots t_i$ by applying $\mathcal{R}$, we start by moving t_{i+1} from left to right, then t_{i+2}, etc.

From Lemma 5.1 we obtain that $\mathcal{R}_c^* \subseteq \mathcal{C}_{\mathcal{R}}$. Since $\mathcal{C}_{\mathcal{R}} \subseteq \mathcal{R}_c^{2|\Sigma|}$, we conclude finally that $\mathcal{R}_c^* = \mathcal{R}_c^{2|\Sigma|}$. □

6 Complexity results

In this section we consider basic complexity questions concerning languages in APC. First, we obtain that both the problem of testing inclusion (or universality) and the problem of deciding whether a language in APC is closed under a semi-commutation

relation are PSPACE-complete. Clearly, these are basic operations when we want to perform model-checking on APC properties. For example, we might ask whether an APC property ϕ_1 is covered by another property ϕ_2, i.e., whether $\phi_1 \subseteq \phi_2$. The test for $\mathcal{R}$-closure is important when we want to know whether a property ϕ is already closed under semi-commutation rewriting, since it avoids computing the $\mathcal{R}$-closed expression which has products of exponential size. Moreover, in the fixed point computations of regular model checking we check whether we have already computed the set of all reachable configurations by an equality test.

For lack of space we omit all proofs of the section and refer to the full version of the paper.

Theorem 6.1 *The following problem is PSPACE-complete:*
Input: *An APC expression L over Σ^*.*
Question: *Is $L = \Sigma^*$?*

Corollary 6.1 *Deciding inclusion for languages in APC is PSPACE-complete.*

Theorem 6.2 *The following problem is PSPACE-complete:*
Input: *An APC expression L over Σ and a semi-commutation rewriting system $\mathcal{R} \subseteq \Sigma \times \Sigma$.*
Question: *Does $\mathcal{R}^*(L) = L$ hold?*

Next, we show that the membership problem for the class APC is PSPACE-complete when we are given a non-deterministic automaton. The same question is NLOGSPACE-complete, hence polynomial, when the input is a deterministic automaton. These two last results rely on the characterization of languages in APC by positive varieties given in [18]. It is worthnoting that the algorithm obtained in [18] has complexity in $O(|A| \cdot 2^{|\Sigma|})$, i.e., it is linear in the size of the automaton and exponential in the size of the alphabet. Theorem 6.4 below improves the result by giving an algorithm which is polynomial in both $|A|$ and $|\Sigma|$.

Theorem 6.3 *Deciding whether a regular language, given by a regular expression or a non-deterministic automaton, is an APC language, is a PSPACE-complete problem.*

Theorem 6.4 *Deciding whether a regular language, given by a deterministic automaton, is an APC language, is an NLOGSPACE-complete problem.*

7 Conclusion

We have identified a class of regular expressions which appears naturally in many contexts, in particular in modeling and verifying concurrent systems and in regular model checking, and we have studied its closure properties and its complexity.

In particular, we have shown that the class of APCs is effectively closed under semi-commutation rewriting (for any such rewriting system). As far as we know, this is the first time that a non-trivial subclass of regular properties has been shown to enjoy this property. As mentioned previously, APCs correspond to level 3/2 in Straubing's concatenation hierarchy, and to level Σ_2 in the quantifier-alternation hierarchy of first-order logic. It is interesting to note that this is the largest class in both hierarchies which is closed under semi-commutation rewriting. However, this raises the question of finding other subclasses of regular languages which have the same closure properties as APC. A minimal requirement on such classes is that Parikh images of their languages should correspond to Presburger formulas where linear constraints do not involve more than one free variable. It can be seen for instance that this property does not hold for $(ab)^*$ whereas it holds for all APC languages.

Another novel contribution of our paper is to show that APCs are also closed under circular semi-commutation rewriting. Actually, our proof holds for any class of languages which is effectively closed under semi-commutation rewriting and conjugacy, since we show that for any system $\mathcal{R}$, computing the circular $\mathcal{R}$-closure reduces to a finite iteration (two times the size of the alphabet) of the computation of the $\mathcal{R}$-closure in alternation with conjugacy. Our result on the closure of APC under semi-commutation rewriting can be applied in modeling and verifying automatically parametrized networks having a ring topology, where information is exchanged between neighbors. Then, an interesting problem is to extend this work to similar systems with other kinds of topologies such as trees and grids.

References

[1] P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded,

lossy fifo channels. In *Proc. of CAV'98*, LNCS 1427, pp. 305–318, 1998.

[2] P. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *Proc. of CAV'99*, LNCS 1633, pp. 134–145, 1999.

[3] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. of CAV'96*, LNCS 1102, pp. 1–12, 1996.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. of CONCUR'97*, LNCS 1243, pp. 135–150, 1997.

[5] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Liveness and acceleration in parametrized verification. In *Proc. of CAV'00*, LNCS 1855, pp. 403–418, 2000.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[8] L. Fribourg and H. Olsén. Reachability sets of parametrized rings as regular languages. *Electronic Notes in Theoretical Computer Science*, pp. 1–12, 1997.

[9] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[10] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. of TACAS 2000*, LNCS 1785, pp. 220–234, 2000.

[11] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proc. of CAV'97*, LNCS 1254, pp. 424–435, 1997.

[12] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[13] A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *Proc. of MFCS'99*, LNCS 1672, pp. 81–91, 1999.

[14] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer, 1991.

[15] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of FoSSaCS'98*, LNCS 1378, pp. 226–242, 1998.

[16] A. Muscholl. Über die Erkennbarkeit unendlicher Spuren. Teubner Verlag, Stuttgart-Leipzig, 1996.

[17] D. Peled. All from one, one from all: on model checking using representatives. In *Proc. of CAV'93*, LNCS 697, pp. 409–423, 1993.

[18] J.-E. Pin and P. Weil. Polynomial closure and unambiguous product. *Theory of Computing Systems*, 30:383–422, 1997.

[19] A. Pnueli and E. Shahar. Liveness and acceleration in parametrized verification. In *Proc. of CAV'00*, LNCS 1855, pp. 328–343, 2000.

[20] D. A. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195(2):183–203, 1998.

[21] W. Thomas. Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25:360–376, 1982.

[22] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.

[23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS '86*, pp. 332–344, 1986.

[24] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, LNCS 1427, pp. 88–97,1998.

Temporal Logic Query Checking (Extended Abstract)

Glenn Bruns Patrice Godefroid
Bell Laboratories, Lucent Technologies
263 Shuman Boulevard, Naperville, IL 60566, U.S.A.
Email: {grb,god}@bell-labs.com

Abstract

A temporal logic query checker takes as input a Kripke structure and a temporal logic formula with a hole, and returns the set of propositional formulas that, when put in the hole, are satisfied by the Kripke structure. By allowing the temporal properties of a system to be discovered, query checking is useful in the study and reverse engineering of systems.

Temporal logic query checking was first proposed in [2]. In this paper, we generalize and simplify Chan's work by showing how a new class of alternating automata can be used for query checking with a wide range of temporal logics.

1 Introduction

As pointed out by Chan in [2], model checking is as often used for understanding a design as for verifying its correctness. One rarely begins the study of a design with a complete specification in hand. Instead, one identifies a few key properties, expresses them in temporal logic, and checks them against the design. Some of the properties usually fail to hold, so the properties (and possibly the design) are revised and rechecked. As this process iterates one develops a more detailed picture of the properties the design satisfies or should satisfy.

To speed the process of design understanding, Chan proposed temporal logic query checking [2]. Here one works with a temporal logic formula containing a placeholder, or hole. A query checker returns the strongest propositional formula that, when put into the hole, is satisfied by the design. For example, given a design and the CTL query $AG?$, the query checker will return the strongest invariant of the system; i.e. the strongest propositional formula that is satisfied in every state of the design. Thus, a query checker allows the mechanization of much of the trial-and-error work done while analyzing a design.

The aim of this paper is to extend and simplify Chan's work. Chan studied CTL query checking, and was interested in queries for which a single strongest solution exists, called *valid* queries in [2]. He showed that it is expensive to determine whether a CTL query is valid, and identified a syntactic class of CTL queries such that every formula in the class is valid. His query-checking algorithm works only with queries in this class. In contrast, we are interested in all CTL queries, even those that have multiple maximally-strong solutions. Furthermore, we do not restrict our attention to CTL. Our query-checking approach is defined for an arbitrary temporal logic.

We simplify Chan's work by showing that query checking can be accomplished by adapting existing model-checking algorithms. In particular, we show how to adapt the automata-theoretic approach to model checking of Kupferman, Vardi and Wolper [8] to solve the query-checking problem.

In the following section of the paper we define the query-checking problem and compare it to model checking. In Section 3, we present some properties of lattices that are central to understanding the solution space of query checking. In Section 4, we outline our approach to query checking and introduce a new class of alternating automata. In Section 5, we show how a query-checking algorithm can be obtained for any logic having a translation to alternating automata, and we describe the application of this approach to CTL queries. In Section 6 we present some examples. Proofs of most theorems are omitted in this extended abstract.

0-7695-1281-X/01 $10.00 © 2001 IEEE

2 Problem Statement

In this section we define the query-checking problem. Our definition is relative to any temporal logic that is interpreted on Kripke structures and that allows atomic propositions as formulas. We write $(K, s) \models \phi$ if state s of Kripke structure K satisfies temporal logic formula ϕ.

A *query* is an expression obtained by replacing a single atomic proposition in a temporal logic formula by the symbol ?, which is referred to as the *placeholder* (or *hole*) of the query. Substituting the placeholder of a temporal logic query by a propositional formula (i.e., a formula built only from atomic propositions and boolean operators) yields a temporal logic formula. We write $\phi[\psi]$ for the formula obtained by substituting propositional formula ψ for the placeholder in query ϕ. We also accept temporal logic formulas themselves as queries. If ϕ is a temporal logic formula, then $\phi[\psi]$ is identical to ϕ.

A propositional formula ψ is a *solution* to a query ϕ, relative to state s of Kripke structure K, if $(K, s) \models \phi[\psi]$.

A *positive* query is a query ϕ that is monotonic with respect to its placeholder: if $\psi_1 \Rightarrow \psi_2$ then $\phi[\psi_1] \Rightarrow \phi[\psi_2]$ (where $\Rightarrow$ denotes logical implication). In what follows we consider only positive queries. With such a query it makes sense to compute only maximally strong solutions, because from these solutions all others can be inferred[1]. Formally, let $PF(P)$ stand for the set of propositional formulas that can be built from a set P of atomic propositions. The ordering $\leq$ on set $PF(P)$ is defined as $\psi_1 \leq \psi_2$ iff $\psi_1 \Rightarrow \psi_2$. The resulting ordered set $\langle PF(P), \leq \rangle$ is a boolean lattice, which we refer to as L_P. For any ordered set $\langle A, \leq \rangle$ and $B \subseteq A$, we define $\min(B)$ by $\{b \in B \mid \forall b' \in B. b' \leq b \Rightarrow b' = b\}$. A subset B of A is *minimal* if $\min(B) = B$.

Definition 1 Let P be a set of atomic propositions, and let P' be a subset of P. Let K be a Kripke structure containing state s, and let ϕ be a query, both defined over P. The *query-checking problem* is to compute the set $\min\{\psi \in PF(P') \mid (K, s) \models \phi[\psi]\}$ of strongest solutions to ϕ. ■

We write $[(K, s), \phi]_{P'}$, or $[(K, s), \phi]$ for short, for the set of strongest solutions to query ψ relative to state s of Kripke structure K and set P' of atomic propositions.

[1] Our restriction to positive queries does not reduce generality. Suppose we had a query with a negated placeholder. We could compute the solution set for this query by removing the negation on the placeholder, computing the solution set for the resulting query, negating each formula in this set, and then interpreting the result as the set of *weakest* solutions to the query.

For a query ϕ without a placeholder, query checking reduces to model checking. If $(K, s) \not\models \phi$, then $(K, s) \not\models \phi[\psi]$ for all propositional formulas ψ, and hence $[(K, s), \phi] = \emptyset$. Otherwise $(K, s) \models \phi$, so $(K, s) \models \phi[\psi]$ for all propositional formulas, and hence $[(K, s), \phi] = \{false\}$. Since query checking is a generalization of model checking, it is at least as hard. Conversely, it is easy to show that query checking itself can be reduced to several model-checking problems.

Theorem 2 *Given a fixed set P' of atomic propositions and a temporal logic TL, the query-checking problem and the model-checking problem for TL have the same complexity in the size of the Kripke structure and in the size of the query/formula.*

Proof: A naive query-checking algorithm for solving $[(K, s), \phi]_{P'}$ consists of enumerating all $L = 2^{2^{|P'|}}$ possible solutions ψ, checking whether $(K, s) \models \phi[\psi]$ for each such ψ, and then returning only the minimal elements from that set. Query checking is thus reduced to at most $2^{2^{|P'|}}$ model-checking problems with a formula of length at most $|\phi| + O(2^{|P'|})$. ■

Since there can be $O(2^{2^{|P'|}})$ minimal solutions to a query-checking problem, parameter P' provides a way to control the complexity of query checking in practice, by specifying the atomic propositions that will appear in solutions computed for the query.

In the remainder of this paper, we develop a constructive algorithm for solving the query-checking problem that can converge directly to its minimal solutions, instead of guessing and checking exponentially-many individual potential solutions one by one as done with the above naive algorithm.

We illustrate the query-checking problem and our ideas to solve it by presenting examples of queries in the temporal logic CTL [5, 10]. Let p range over a set P of atomic propositions. The abstract syntax of CTL is defined from *state formulas* ϕ and *path formulas* ψ as follows:

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid A\psi \mid E\psi$$
$$\psi ::= X\phi \mid \phi_1 \, \mathcal{U} \, \phi_2 \mid \phi_1 \, \tilde{\mathcal{U}} \, \phi_2$$

A CTL formula is a state formula. The *closure* of a CTL formula ϕ, written $cl(\phi)$, is defined as the set of all state subformulas of ϕ. The *size* $|\phi|$ of a formula ϕ is defined as the number of elements of $cl(\phi)$.

A CTL formula is interpreted with respect to a *Kripke structure* $K = (P, S, s_0, R, L)$ where P is a finite set of atomic propositions, S is a finite set of states, s_0 in S is the initial state, $R \subseteq S \times S$ is a total transition relation on states, and $L : S \to 2^P$ is a labeling

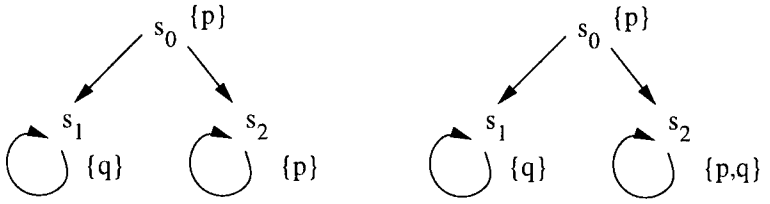


Figure 1. Example Kripke structures K_1 and K_2

function that maps each state to a set of atomic propositions. A *path* $w = s_0, s_1, \ldots$ of a Kripke structure is an infinite sequence of states such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. We write w^i for the ith state of path w, with w^0 the first state. Also, we write paths(s) for the set of all paths w in K such that w^0 is s.

Given a Kripke structure $K = (P, S, s_0, R, L)$, a state formula ϕ satisfies a state s of K, and a path formula ψ satisfies a path w of K, according to the following inductive definitions.

$$
\begin{aligned}
(K,s) \models p &\stackrel{\text{def}}{=} p \in L(s) \\
(K,s) \models \neg p &\stackrel{\text{def}}{=} p \notin L(s) \\
(K,s) \models \phi_1 \wedge \phi_2 &\stackrel{\text{def}}{=} (K,s) \models \phi_1 \text{ and } (K,s) \models \phi_2 \\
(K,s) \models \phi_1 \vee \phi_2 &\stackrel{\text{def}}{=} (K,s) \models \phi_1 \text{ or } (K,s) \models \phi_2 \\
(K,s) \models A\psi &\stackrel{\text{def}}{=} \forall w \in \text{paths}(s).(K,w) \models \psi \\
(K,s) \models E\psi &\stackrel{\text{def}}{=} \exists w \in \text{paths}(s).(K,w) \models \psi \\
\\
(K,w) \models X\phi &\stackrel{\text{def}}{=} (K,w^1) \models \phi \\
(K,w) \models \phi_1 \,\mathcal{U}\, \phi_2 &\stackrel{\text{def}}{=} \exists i.(K,w^i) \models \phi_2 \text{ and} \\
&\qquad \forall j < i.(K,w^j) \models \phi_1 \\
(K,w) \models \phi_1 \,\tilde{\mathcal{U}}\, \phi_2 &\stackrel{\text{def}}{=} \forall i.(K,w^i) \models \phi_2 \text{ or} \\
&\qquad \exists j < i.(K,w^j) \models \phi_1
\end{aligned}
$$

The class of CTL queries we allow are those for which negation is not applied to the placeholder. All such queries are positive.

Consider the CTL query $A(\mathit{false}\,\tilde{\mathcal{U}}\,?)$ (sometimes written $AG?$) and Kripke structure K_1, which is shown on the left of Figure 1. The formula $A(\mathit{false}\,\tilde{\mathcal{U}}\,\phi)$ holds if formula ϕ holds everywhere along all paths of a structure. A solution to the query is therefore a maximally-strong propositional formula that holds everywhere in the Kripke structure. Informally, the strongest solution of $\mathit{true}\,\mathcal{U}\,?$ for the left path in the example is $p \neq q$, and strongest solution for the right path is $p \wedge \neg q$. Therefore, the strongest solution that holds for all paths is $p \neq q$.

Consider the same query and Kripke structure K_2. Here the strongest solution on the left branch is $p \neq q$ and the strongest solution on the right branch is p. The strongest solution for both paths is therefore $p \vee q$.

Now, consider the CTL query $E(\mathit{true}\,\mathcal{U}\,?)$ (sometimes written $EF?$) and Kripke structure K_1. A solution to this query is a maximally-strong propositional formula that holds anywhere in the Kripke structure. This query on K_1 has two maximally-strong solutions: $p \wedge \neg q$ and $q \wedge \neg p$. The same query evaluated on K_2 has three maximally-strong solutions: $p \wedge \neg q$, $q \wedge \neg p$, and $p \wedge q$.

3 Solutions to Queries

In model checking with alternating automata, conjunction and disjunction operations are performed on truth values. In our algorithm for query checking, analogous operations are performed on sets of maximally-strong propositional formulas. These operations are defined as the meet and join operations of a lattice. In this section we define this lattice and show properties of the meet and join operations.

To begin, recall that we write $\psi_1 \leq \psi_2$ for propositional formulas ψ_1 and ψ_2 if $\psi_1 \Rightarrow \psi_2$. Also, given a set P of atomic propositions we write L_P for the boolean lattice $\langle PF(P), \leq \rangle$ having as its elements the propositional formulas built from elements of P. The left-most lattice of Figure 2 is L_P, where P contains only the single atomic proposition p.

Before going directly to the definition of a lattice on sets of maximally-strong propositional formulas, we will define a related lattice. Consider the set of *all* solutions to a query, not just the minimal ones. Because our queries are positive, the set of all solutions to a query is a set of propositional formulas that is closed under "going-up" with respect to $\leq$. In other words, if some propositional formula belongs to the the set, then so does every weaker formula. Given an ordered set $\langle A, \leq \rangle$ and a subset B of A, we define

$$\uparrow B \stackrel{\text{def}}{=} \{a \in A \mid \exists b \in B.b \leq a\}$$

A subset B of A is an *up-set* if $\uparrow B = B$. We write $U(A)$ for the set of all up-sets of A. Lattice theory (see

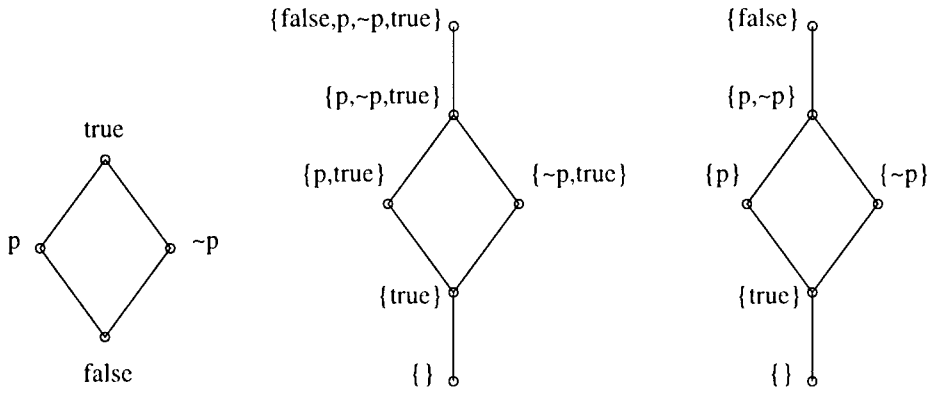


Figure 2. Lattices L_P, $L_P^{\uparrow}$, and L_P^{min} for $P = \{p\}$

Sec. 8.20 of [6]) tells us that if A is finite then $U(A)$ is a finite, distributive lattice, with elements ordered by set inclusion. It is easy to see that the meet and join operations of $U(A)$ are just set intersection and union.

Given a set P of atomic propositions, let $L_P^{\uparrow}$ be the lattice $U(PF(P))$, which is finite and distributive, but not boolean (see Lemma 8.21 of [6]). The middle lattice of Figure 2 is $L_P^{\uparrow}$ for $P = \{p\}$. Each element of this lattice is a possible set of solutions to a query in which the set of atomic propositions contains only atomic proposition p. Although not evident from Figure 2, lattice $L_P^{\uparrow}$ grows much faster than L_P as the set P of atomic propositions grows.

Each element of $L_P^{\uparrow}$ can be represented by its minimal elements. Recall from Section 2 that $\min(A)$ stands for the minimal elements of an ordered set A.

Proposition 3 *Let $\langle A, \leq \rangle$ be an ordered set with $B, C \subseteq A$. Then*

$$\begin{aligned} min(\uparrow B) &= min(B) \\ min(B \cup C) &= min(min(B) \cup min(C)) \end{aligned}$$

From $L_P^{\uparrow}$ we get an isomorphic lattice L_P^{min} by applying min to each element. Each element of L_P^{min} represents a set of maximally-strong propositional formulas, i.e., a candidate set of solutions to a query. The ordering of L_P^{min} is derived from the ordering of $L_P^{\uparrow}$: $A \leq B$ in $L_P^{\uparrow}$ if $\uparrow A \subseteq \uparrow B$. Similarly, the meet and join operations of L_P^{min} (which we write as $\underline{\wedge}$ and $\underline{\vee}$) are derived from $L_P^{\uparrow}$:

$$\begin{aligned} A \mathbin{\underline{\wedge}} B &\stackrel{\text{def}}{=} \min(\uparrow A \cap \uparrow B) \\ A \mathbin{\underline{\vee}} B &\stackrel{\text{def}}{=} \min(\uparrow A \cup \uparrow B) \end{aligned}$$

The right-most lattice of Figure 2 is L_P^{min} for $P = \{p\}$.

Defining $\underline{\wedge}$ and $\underline{\vee}$ as the meet and join operations of a distributive lattice is helpful because we immediately learn some properties of $\underline{\wedge}$ and $\underline{\vee}$.

Proposition 4 *Let A, B, and C be elements of L_P^{min}. Then*

$$\begin{aligned} A \mathbin{\underline{\wedge}} B &= B \mathbin{\underline{\wedge}} A \\ A \mathbin{\underline{\vee}} B &= B \mathbin{\underline{\vee}} A \\ A \mathbin{\underline{\wedge}} (B \mathbin{\underline{\wedge}} C) &= (A \mathbin{\underline{\wedge}} B) \mathbin{\underline{\wedge}} C \\ A \mathbin{\underline{\vee}} (B \mathbin{\underline{\vee}} C) &= (A \mathbin{\underline{\vee}} B) \mathbin{\underline{\vee}} C \\ A \mathbin{\underline{\wedge}} (B \mathbin{\underline{\vee}} C) &= (A \mathbin{\underline{\wedge}} B) \mathbin{\underline{\vee}} (A \mathbin{\underline{\wedge}} C) \\ A \mathbin{\underline{\vee}} (B \mathbin{\underline{\wedge}} C) &= (A \mathbin{\underline{\vee}} B) \mathbin{\underline{\wedge}} (A \mathbin{\underline{\vee}} C) \end{aligned}$$

It is awkward to compute $A \mathbin{\underline{\wedge}} B$ and $A \mathbin{\underline{\vee}} B$ using the definitions of $\underline{\wedge}$ and $\underline{\vee}$ directly because they first expand A and B to $\uparrow A$ and $\uparrow B$. The following characterizations allow $\underline{\wedge}$ and $\underline{\vee}$ to be computed directly using minimal sets.

Theorem 5 *Let A and B be elements of L_P^{min}. Then*

$$\begin{aligned} A \mathbin{\underline{\wedge}} B &= min(\{a \vee b \mid a \in A \text{ and } b \in B\}) \\ A \mathbin{\underline{\vee}} B &= min(A \cup B) \end{aligned}$$

4 Extended Alternating Automata

Inspired by the automata-theoretic approach to model checking of [8], we propose the following approach to query checking. Given a temporal logic query ϕ and a Kripke structure K, we (1) build an alternating automaton representing ϕ, (2) compute the product of this automaton with K, and finally (3) check whether the language accepted by the product automaton is empty. A key step in developing this approach is to discover a kind of alternating automaton appropriate for representing a temporal logic query. In this section we introduce a new type of alternating automata for this purpose, which we call *extended alternating automata* (EAA).

The novel aspect of alternating automata [3] is that the transition function maps an automaton state and

x_3 respectively on the right-hand side are replaced by $\{false\}$, since the state is accepting. Then, the values for x_2 and x_3 are computed by applying the definition of $\underline{\wedge}$: one obtains $x_2 = \{q \wedge \neg p\}$ and $x_3 = \{p \wedge \neg q\}$. The algorithm then backs up to x_1 and computes the value of x_1, which is $\{p \neq q\}$. This value is the solution to the query $[(K_1, s_0), A(false\, \tilde{\mathcal{U}}\, ?)]$.

7 Discussion

We have presented a general automata-theoretic approach to temporal logic query checking. The approach is general in the sense that if one has a translation from queries to EAA in the sense of Theorem 7, then checking nonemptiness of the product automaton gives the solution to the query. For CTL we showed how this translation can be derived directly from the translation of CTL to alternating automata. Translations for queries in other temporal logics (such as the modal mu-calculus) can be derived similarly.

We have defined EAA relative to an arbitrary finite lattice, although for query checking we need only EAA based on a lattices of the form L_P^{min}. A general definition for EAA was chosen because it is simpler, and also because we can imagine other uses for the more general form. For example, EAA could be used for model checking multi-valued temporal logics [7, 1, 4].

References

[1] G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *Proceedings of the 11th Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287, Trento, July 1999. Springer-Verlag.

[2] W. Chan. Temporal-Logic Queries. In *Proceedings of the 12th Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463, Chicago, July 2000. Springer-Verlag.

[3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[4] M. Chechik, W. Easterbrook, and V. Petrovykh. Model-Checking over Multi-Valued Logics. In *Proceedings of FME '01*, pages 72–98, March 2001.

[5] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Jan. 1986.

[6] B. Davey and H. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[7] M. Fitting. Many-valued modal logics I. *Fundamenta Informaticae*, 15:235–254, 1992.

[8] O. Kupferman, M. Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[9] D. Muller, A. Saoudi, and P. Schupp. Alternating automata and the weak monadic theory of the tree and its complexity. In *Proceedings of ICALP '86, Lecture Notes in Computer Science*, pages 275–283. Springer-Verlag, 1986.

[10] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

Session 11

Typechecking XML Views of Relational Databases*

Noga Alon
Tel Aviv University
noga@tau.math.ac.il

Tova Milo
Tel Aviv University
milo@tau.math.ac.il

Frank Neven†
Limburgs Universitair Centrum
frank.neven@luc.ac.be

Dan Suciu
University of Washington
suciu@cs.washington.edu

Victor Vianu‡
U.C. San Diego
vianu@cs.ucsd.edu

Abstract

Motivated by the need to export relational databases as XML data in the context of the Web, we investigate the typechecking *problem for transformations of relational data into tree data (XML). The problem consists of statically verifying that the output of every transformation belongs to a given output tree language (specified for XML by a DTD), for input databases satisfying given integrity constraints. The typechecking problem is parameterized by the class of formulas defining the transformation, the class of output tree languages, and the class of integrity constraints. While undecidable in its most general formulation, the typechecking problem has many special cases of practical interest that turn out to be decidable. The main contribution of this paper is to trace a fairly tight boundary of decidability for typechecking in this framework. In the decidable cases we examine the complexity, and show lower and upper bounds. We also exhibit a practically appealing restriction for which typechecking is in* PTIME.

1 Introduction

Since Codd [8], databases have been modeled as first-order relational structures and database queries as mappings from relational structures to relational structures. This captured well relational databases, where both data and query answers are represented as tables.

Today's technology trends require us to model data that is no longer tabular. The World Wide Web Consortium has adopted a standard data exchange format for the Web, called Extended Markup Language (XML) (see [1]), in which data is represented as a labeled ordered tree, rather than as a table. XML is rapidly becoming the de facto data format on the Web, and many industries (e.g. financial, manufacturing, health care) are migrating their application-specific formats to XML. All major database vendors offer now tools for exporting relational data as XML, thus making it easier for companies to define XML views of their relational data and share it with business partners over the Web. An important aspect of XML is that it allows users to define *types*. A type is a tree language, and the current standards for XML types (DTD and XML-Schema) correspond to restricted regular tree languages. XML data exchange is always done in the context of a fixed type: a community (or industry) agrees on a certain type, and subsequently all members of the community create XML views of their relational data that are of that type.

In this paper we study the problem of mapping relational data into tree data, specifically addressing the *typechecking* problem. Given a mapping and a type for the output tree, we wish to automatically check whether every database is mapped to a tree of the desired output type. As explained, this is a critical problem in XML data exchange. In addition, as we show here, this problem is also technically interesting and non-trivial from a theoretical perspective.

We define a language, TreeQL, expressing mappings from relational structures to trees. A mapping m in TreeQL is specified as a tree where each node is labeled by a logical formula, possibly with free variables, and a symbol from a finite alphabet Σ. An ordered relational structure is mapped into a Σ-tree whose nodes consists of all tuples that satisfy some formula in the tree, and whose edges are defined based on the edges in m. In the typechecking problem we are given a regular tree language, called the *output type*, and a set of integrity constraints, and are

*Work supported in part by the U.S.-Israel Binational Science Foundation under grant number 97-00128.

†Post-doctoral researcher of the Fund for Scientific Research, Flanders.

‡This author supported in part by the National Science Foundation under grant number IIS-9802288.

0-7695-1281-X/01 $10.00 © 2001 IEEE

asked to check whether every input structure satisfying the constraints is mapped into a tree in the output type. Solving the typechecking problem boils down to checking whether the strings generated by the ordered sets of tuples satisfying a sequence of logical formulas belong to some regular language. The typechecking problem is parameterized by the fragment of TreeQL, the class of output types, and the class of integrity constraints.

The typechecking problem in its various instantiations requires an understanding of the interaction between logic and tree languages. We found this interaction interesting, and had to develop distinct approaches for the different instances of the typechecking problem, combining techniques from finite-model theory, language theory, and combinatorics.

It is easily seen that typechecking becomes undecidable when arbitrary first-order logic (FO) formulas are allowed in the mapping, due to a reduction from the FO finite satisfiability problem. Hence, we focus our investigation on the particular case when the formulas are *conjunctive queries*. When the output types are further restricted to *star-free* regular languages, typechecking is decidable. When the output type is an arbitrary regular expression, typechecking is still decidable for *projection-free* conjunctive formulas (the proof uses a combinatorial argument based on Ramsey's theorem). On the other hand, we show that even small extensions to the basic decidable cases lead to undecidability of typechecking. Thus, our results provide a fairly tight boundary of decidability of typechecking. A side benefit is new insight into the subtle interplay between constraints, query languages, and output tree types.

Related work. Type inference is a well-studied topic in functional programming languages [15]. A type inference system consists of a set of inference rules that can be used to check whether a function (program) is type safe. This means that during execution the program will never get into a state where it attempts to apply an operator to operands of wrong types. The problem we consider here is differentWe are checking a semantic property, namely whether every input database is mapped to an output tree of the right type, which is in contrast to the syntactic nature of applying the type inference rules. In our setting type checking rapidly becomes undecidable if we allow the transformation language or the output types to be too expressive. In contrast, type inference for functional programming languages (that are Turing complete) is usually decidable for powerful type systems but is only sound.

Our work is motivated by the practical need to typecheck XML views from relational databases. SilkRoute [10] is a research prototype enabling an XML view to be defined from a relational database using a declarative language. The language TreeQL used in the present paper is an abstraction of the language used by SilkRoute.

A different but related problem is that of typechecking tree transformations. In previous work [14] a subset of the authors studied the typechecking problem for transformations of unranked trees expressed by k-pebble transducers, and showed that typechecking is decidable. The unranked trees considered there are labeled over a fixed, finite alphabet Σ. So they do not take into account the data values present in XML documents. In subsequent work [3] we considered trees with labels from an infinite alphabet, that model more closely XML trees where internal nodes have labels from a known, fixed alphabet, while leaves contain data values from an infinite domain. We showed that typechecking quickly becomes undecidable, even if one considers very restricted transformations. However, typechecking becomes decidable for several restrictions on the class of transformations and/or the tree types. While some of the techniques in [3] are similar in flavor to those in the present paper, there are considerable differences in the two settings. Relational structures can be encoded as XML, but the integrity constraints do not have an analog in XML. Conversely, the DTDs that constrain XML documents cannot be expressed by the relational constraints we consider. However, some of the lower bound results in the present paper can be transferred to the XML context and strengthen results from [3]. A more detailed comparison is deferred to the full version of this paper.

Organization The paper is organized as follows. The first section develops the basic formalism, including our abstraction of XML documents, DTDs, and the variant of TreeQL used as transformation language. Section 3 presents the decidability results; Section 4 the complexity analysis; and Section 5 the undecidability results. The paper ends with brief conclusions. Due to space limitations, some proofs are only sketched or omitted entirely.

2 Basic Framework

We introduce here the basic formalism used throughout the paper, including our abstraction of XML documents, DTDs, and the query language TreeQL.

Trees. Trees are our abstraction of XML documents [1]. They capture the nesting structure of XML elements and their tags. We refrain from modeling data values as they are not relevant w.r.t. typechecking. Indeed, output types only constrain the struc-

ture of the output tree not the data values at the leaves. We consider ordered trees with node labels from a finite alphabet Σ. We also refer to such trees as Σ-trees. We denote by nodes(t) the set of *nodes* of a tree t; for a node v, we denote by lab(v) the *label* of v. There is no a priori bound on the number of children of a node; we therefore call these trees *unranked*. We denote the empty tree by ε and the set of all trees over Σ by $\mathcal{T}_\Sigma$. By root(t), we denote the root of t. To define the semantics of TreeQL programs we also need the notion of a *forest* which is just a sequence of trees. We employ the following notational convenience. By $\sigma(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are trees, we mean the tree where the root is labeled with σ and the i-th subtree is t_i.

Types and DTDs. DTDs and their variants provide a typing mechanism for XML documents. We use several notions of types for trees. For $\mathcal{C}$ a class of string languages over Σ, a *DTD over Σ w.r.t. $\mathcal{C}$* is a mapping from Σ to languages in $\mathcal{C}$. We denote the class of all such DTDs by DTD($\mathcal{C}$). Let $d \in$ DTD($\mathcal{C}$). Then, a Σ-tree t *satisfies* d, if for every node v of t with children $v_1, \ldots, v_n$, $\text{lab}(v_1) \cdots \text{lab}(v_n) \in d(\text{lab}(v))$. Note that, if $n = 0$, then ε should belong to $d(\text{lab}(v))$. The set of trees that satisfy d is denoted by $L(d)$.

Obvious examples of classes $\mathcal{C}$ are the regular languages (REG), the star-free regular languages (SF), and the context-free languages (CFL). When $\mathcal{C}$ are the regular languages our notion of DTDs corresponds closely to the DTDs proposed for XML documents. Star-free regular languages are defined by the *star-free* regular expressions, which are build from single symbols and ε, using concatenation, union, and complement. They correspond exactly to the languages defined by first-order logic (FO) over the vocabulary $\{<, (O_\sigma)_{\sigma\in\Sigma}\}$ where $<$ is a binary relation and every O_σ is a unary relation [13, 18]. A string $w = \sigma_1 \ldots \sigma_n$ is then represented by the logical structure $(\{1, \ldots, n\}; <, (O_\sigma)_{\sigma\in\Sigma})$ where $<$ is the natural order on $\{1, \ldots, n\}$, and for each i, $i \in O_\sigma$ iff $\sigma_i = \sigma$.

We will consider an even simpler class of DTDs, which specify cardinality constraints on the tags of children of a node, but does not restrict their order. Such DTDs are useful either when order is irrelevant, or when the order of tags in the output is hard-wired by the syntax of the query and so can be factored out. We use a logic called $\mathcal{SL}$, inspired by [16]. The syntax of the language is as follows. For every $\sigma \in \Sigma$ and natural number i, $\sigma^{=i}$ and $\sigma^{\geq i}$ are *atomic $\mathcal{SL}$ formulas*; true is also an atomic $\mathcal{SL}$ formula. Every atomic formula is a formula and the negation, conjunction, and disjunction of formulas are also formulas. A string w over Σ satisfies an atomic formula $\sigma^{=i}$ if it has exactly i occurrences of σ, and similarly for $\sigma^{\geq i}$. Further, true is satisfied by every string. [1] Satisfaction of Boolean combination of atomic formulas is defined in the obvious way. As an example, consider the $\mathcal{SL}$ formula co-producer$^{\geq 1}$ $\rightarrow$ producer$^{\geq 1}$. This expresses the constraint that a co-producer can only occur when a producer occurs. One can check that languages expressed in $\mathcal{SL}$ correspond precisely to properties of structures over the vocabulary $\{<, (O_\sigma)_{\sigma\in\Sigma}\}$ that can be expressed in FO without using the order relation, $<$. Thus, $\mathcal{SL}$ forms a natural subclass of the star-free regular expressions.

We have so far defined DTDs and several restrictions. We next consider an orthogonal *extension* of basic DTDs, also present in more recent DTD proposals such as XML-Schemas [4, 5]. This is motivated by a severe limitation of basic DTDs: their definition of the type of a given tag depends only on the tag itself and not on the context in which it occurs. For example, this means that the singleton $\{t\}$ where t is the tree $a(b(c), b(d))$ cannot be described by a DTD, because the "type" of the first b differs from that of the second b. This naturally leads to an extension of DTDs with *specialization* (also called decoupled types) which, intuitively, allows defining the type of a tag by several "cases" depending on the context. Formally, we have:

Definition 2.1. *For a class of languages $\mathcal{C}$, a* specialized *DTD over Σ w.r.t. $\mathcal{C}$ is a tuple $\tau = (\Sigma, \Sigma', d, \mu)$ where (i) Σ and Σ' are finite alphabets; (ii) d is a DTD over Σ' w.r.t. $\mathcal{C}$; and (iii) μ is a mapping from Σ' to Σ. A tree t over Σ satisfies a specialized DTD τ, if $t \in \mu(L(d))$. We denote the set of all such specialized DTDs by S-DTD($\mathcal{C}$).*

Intuitively, Σ' provides for some a's in Σ a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$. We also denote by μ the homomorphism induced on strings and trees. Interestingly, it turns out that the class S-DTD(REG) is precisely equivalent to the class of regular tree automata over unranked trees [7, 17]. This is more evidence that specialized DTDs are a robust and natural specification mechanism.

Logic. Consider some fixed relational vocabulary $\mathcal{S}$. A database over $\mathcal{S}$ is just an $\mathcal{S}$-structure defined in the usual way [2, 9]. We denote the domain of a database $\mathcal{A}$ by dom($\mathcal{A}$). Further, let $\mathcal{L}$ be a logic over $\mathcal{S}$. Then we denote the free variables occurring in $\varphi \in \mathcal{L}$ by Free(φ). In the sequel, $\mathcal{L}$ will usually be the set of conjunctive queries over $\mathcal{S}$, denoted by

[1] The empty string is obtained by $\bigwedge_{\sigma\in\Sigma} \sigma^{=0}$ and the empty set by $\neg$true. We, hence, use ε and $\emptyset$ as shorthands in $\mathcal{SL}$ formulas.

CQ. Formally, a conjunctive query is a positive existential first-order logic formula $\varphi(x_1, \ldots, x_n)$ having conjunctions as its only Boolean connective, that is, a formula of the form $\exists y_1 \cdots \exists y_m \psi(\bar{y}, \bar{x})$, where ψ is a conjunction of atomic formulas over $\mathcal{S}$ (so, no equalities). By CQ with superscripts in $\{=, \neg\}$ we mean CQ where ψ can contain equality and negations of atomic formulas, respectively. A conjunctive query is *projection-free* when there are no leading existential quantifiers. Another logic frequently referred to in the sequel consists of the FO formulas of the form $\exists \bar{x} \forall \bar{y} \varphi(\bar{x}, \bar{y})$ with φ quantifier-free. We denote this class by FO($\exists^* \forall^*$).

In relational databases, one usually considers databases satisfying some integrity constraints [2]. These are sentences in a specific logic. A database $\mathcal{A}$ satisfies a set of constraints Φ, if $\mathcal{A} \models \varphi$ for every $\varphi \in \Phi$. We mainly consider constraints specified in FO($\exists^* \forall^*$). Note that they encompass functional dependencies (FDs), but not, for instance, inclusion dependencies (IDs). Recall that FDs are expressions of the form $X \to Y$ where X and Y are sets of coordinates of a relation, and $X \to Y$ holds in a relation if whenever two tuples agree on X they also agree on Y. IDs are of the form $R[i_1, \ldots, i_k] \subseteq S[j_1, \ldots, j_k]$ where R and S are relation symbols, and $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ are natural numbers less than or equal to the arity of R and S, respectively. A database satisfies the above inclusion dependency iff $\pi_{i_1, \ldots, i_k}(R) \subseteq \pi_{j_1, \ldots, j_k}(S)$ where π denotes projection as usual. An inclusion dependency is *unary* when $k = 1$. A set Φ of dependencies is *cyclic* iff either one of the following holds

- Φ contains a dependency of the form $R[\bar{i}] \subseteq R[\bar{j}]$ with $\bar{i} \neq \bar{j}$; or
- Φ contains dependencies $R_1[\bar{i}_1] \subseteq R_2[\bar{j}_2]$, $R_2[\bar{i}_2] \subseteq R_3[\bar{j}_3]$, $\ldots$, $R_m[\bar{i}_m] \subseteq R_1[\bar{j}_1]$.

A set of dependencies is *acyclic* when it is not cyclic. We denote the class of acyclic inclusion dependencies by AcIDs.

Finally, we recall the following technical notion. For a finite set of variables X, an *X-substitution θ for $\mathcal{A}$* is a mapping from X to $\mathrm{dom}(\mathcal{A})$. Let $\bar{x}$ be variables not occurring in X and let $\bar{a}$ be as many elements of $\mathrm{dom}(\mathcal{A})$. Then $\theta \cup \{\bar{x} \mapsto \bar{a}\}$ denotes the $(X \cup \{\bar{x}\})$-substitution that maps each x_i to a_i and every $y \in X$ to $\theta(y)$.

TreeQL. The transformation language we consider, mapping databases to trees, is an abstraction of RXL [10]. We refer to it as TreeQL. The queries are tree patterns where nodes are labeled with label-formula pairs. Therefore, denote by $\Sigma \times \mathcal{L}$ the set of pairs $(\sigma, \varphi(\bar{x}))$ with $\sigma \in \Sigma$, and $\varphi(\bar{x})$ a formula in $\mathcal{L}$. TreeQL programs are trees in $\mathcal{T}_{\Sigma \times \mathcal{L}}$. In the next definition, denote by formula(v) the formula associated to a node v.

Definition 2.2. *A* TreeQL($\mathcal{L}, \Sigma$) program *is a tree $P \in \mathcal{T}_{\Sigma \times \mathcal{L}}$ such that Free(formula(v)) $\subseteq$ Free(formula(v')), for all nodes v and v' where v' is a descendant of v; in addition, the formula in the label of the root is equivalent to true.*

If $\mathcal{L}$ or Σ are clear from the context or not important, we sometimes omit them. Sometimes, we abbreviate the label (σ, true) simply by σ.

Let $\mathcal{A}$ be a database over $\mathcal{S}$, $<$ a total order on $\mathrm{dom}(\mathcal{A})$, and P a TreeQL program.

Definition 2.3. *The tree $P(\mathcal{A}, <)$ generated by P from $\mathcal{A}$ and $<$ is defined as follows. Its nodes consist of pairs of the form (v, θ) where v is a node of P and θ an $\bar{x}$-substitution (where $\bar{x}$ = Free(formula(v))) such that $\mathcal{A} \models \varphi[\theta]$ for every formula φ labeling v or labeling an ancestor of v in P. The root is $(root(P), ())$ and nodes are ordered component-wise, using the node order in P for v and the lexicographic order $<$ on θ. The edges in $P(\mathcal{A}, <)$ are $((v, \theta), (v', \theta'))$ such that v' is a child of v in P and θ' is an extension of θ. Finally the label of a node (v, θ) is the Σ label of v in P.*

Example 2.4. *Consider the TreeQL(CQ) program $P = v_0(v_1, v_2, v_3)$ (i.e. the tree has root node v_0 with children v_1, v_2, v_3) and $lab(v_0) = (a, true)$, $lab(v_1) = (b, R(x, y) \wedge R(y, x))$, $lab(v_2) = (c, R(x, y))$, $lab(v_3) = (d, R(x, y) \wedge R(u, v))$, and consider database $\mathcal{A}$ in which $R = \{(i, j) \mid 0 \leq i \leq j \leq 9\}$, and the natural order $<$ on $\{0, \ldots, 9\}$. Then $P(\mathcal{A}, <)$ is a tree whose root has 10 children labeled b followed by 55 children labeled c and followed by $55^2 = 3025$ children labeled d.*

We remark that RXL [10], the language TreeQL is an abstraction of, also allows to output data values occurring in the input database as labels of leaves in XML documents. However, as we study typechecking and output types do not constrain these data values we chose to omit them from the formalism.

An extension: TreeQL with virtual nodes. We will use an extension of TreeQL that allows programs to define "temporary" nodes, called *virtual*, that are eliminated in the final answer. To see why this is useful, consider an input binary relation R providing titles and speakers of talks (ordered alphabetically by title). Suppose we wish to output a tree listing under the root the ordered title/speaker pairs. This cannot be defined by a TreeQL program, because it cannot group the titles and speakers as required.

However, suppose we can use temporary nodes, identified by a special label #. Consider the query $root((\#, R(t,s))((title, R(s,t)), (speaker, R(s,t))))$. This produces one node labeled # for each tuple in R, whose children are the corresponding title and speaker. The ordered sequence of title/speaker pairs can now be obtained by a "flattening" operation that eliminates the # nodes and concatenates their children.

More formally, let # be a special symbol not occurring in Σ. We denote by $\Sigma_\#$ the set $\Sigma \cup \{\#\}$. The symbol # will be used to specify virtual nodes. Define the function $\lambda_\#$ which maps trees to forests by eliminating #-labeled nodes, recursively as follows. Let t be the tree $\sigma(t_1, \ldots, t_n)$. Then

$$\lambda_\#(t) := \begin{cases} \sigma(\lambda_\#(t_1), \ldots, \lambda_\#(t_n)) & \text{if } \sigma \neq \#; \\ \lambda_\#(t_1), \ldots, \lambda_\#(t_n) & \text{if } \sigma = \#. \end{cases}$$

Definition 2.5. A TreeQL($\mathcal{L}, \Sigma$) *program P with virtual nodes* is a TreeQL($\mathcal{L}, \Sigma_\#$) program where $\text{lab}(\text{root}(P)) \notin \{\#\} \times \mathcal{L}$. We denote the set of all such programs by TreeQL$^{\text{virt}}(\mathcal{L}, \Sigma)$. The tree generated by P from $\mathcal{A}$ and $<$ is defined as $\lambda_\#(P(\mathcal{A}, <))$, and denoted, by slight abuse of notation, also by $P(\mathcal{A}, <)$.

Typechecking. We next formalize the central problem of this paper.

Definition 2.6. *A TreeQL program P* typechecks *with respect to a set of constraints Φ and an output type d iff $P(\mathcal{A}, <) \subseteq L(d)$ for every database $\mathcal{A}$ that satisfies Φ and every total order $<$ on $dom(\mathcal{A})$.*

Example 2.7. *Continuing with Example 2.4, consider the DTD defined by the mapping $d : \{a, b, c, d\} \to REG$ given by:*

$$d(a) = (b^*.(c.c)^*.(d.d)^*) \mid (b^*.(c.c)^*.c.(d.d)^*.d)$$

and $d(b) = d(c) = d(d) = \varepsilon$. The type says that there are an even number of c's and d's or an odd number of both under nodes labeled a. Then the TreeQL program P in Example 2.4 typechecks w.r.t. this DTD.

The typechecking problem is parameterized by (1) the fragment of TreeQL; (2) the output type; and (3) the integrity constraints. Therefore, we denote by

$$\text{TC}[\mathcal{R}, \mathcal{D}, \mathcal{IC}],$$

the above decision problem where $\mathcal{R}$ is a fragment of TreeQL or TreeQL$^{\text{virt}}$, $\mathcal{D}$ is a class of output types, and $\mathcal{IC}$ is a class of integrity constraints. To reduce notation, we abbreviate TreeQL($\mathcal{L}$) and TreeQL$^{\text{virt}}(\mathcal{L})$ by $\mathcal{L}$ and $\mathcal{L}_{\text{virt}}$, respectively; and, we abbreviate DTD($\mathcal{C}$) and S-DTD($\mathcal{C}$) by $\mathcal{C}$ and $\mathcal{C}_{\text{spec}}$, respectively.

Clearly, TC[$\mathcal{L}$, $\mathcal{D}$, $\mathcal{IC}$] is undecidable for any logic $\mathcal{L}$ for which satisfiability is undecidable. Indeed, for a sentence $\varphi \in \mathcal{L}$, consider the program result$((a, \varphi))$ with an output type d that maps d(result) to $\{\varepsilon\}$. Then φ is satisfiable iff the program does not typecheck w.r.t. d.

In the sequel we focus on conjunctive queries, which correspond to the widely used select-project-join queries in SQL. As shown in Section 5, the typechecking problem quickly becomes undecidable. Nevertheless, as shown in the next section, we obtain decidability and even tractability for a large class of transformations.

3 Decidability

We present in this section our decidability results on typechecking TreeQL queries:

(i) When restricting output DTDs to star-free languages we show that typechecking is decidable for TreeQL(CQ$^{=,\neg}$) programs and integrity constraints in FO($\exists^*\forall^*$). The proof gives a CONEXPTIME upper bound. In Section 4, we provide the matching lower bound.

(ii) By restricting the queries to projection-free CQs and the integrity constraints to FDs, we show that typechecking w.r.t. DTDs with full regular expressions is decidable. The proof is based on Ramsey theory and yields a non-elementary upper bound. It is open whether this can be improved.

In Section 5, we show that the above results are essentially optimal: slight increase of the power of the DTDs or the integrity constraints lead to undecidability. However, it remains open whether in (ii) above, the restriction to projection-free CQs is required. We first consider star-free output types and integrity constraints in FO($\exists^*\forall^*$).

Theorem 3.1. TC[CQ$^{=,\neg}$, SF, FO($\exists^*\forall^*$)] *is in* CONEXPTIME.

Proof. The decidability is shown by bounding the size of inputs that need to be checked to detect a violation of the output DTD. Let R be a TreeQL(CQ$^{=,\neg}$) program, let $d \in$ DTD(SF), and let Φ be a finite set of FO($\exists^*\forall^*$) sentences.

We start by stating a technical lemma. Extend the star-free regular expressions by the constructs $\sigma^{=i}$ and $\sigma^{\geq i}$. These denote the languages $\{\sigma^i\}$ and $\{\sigma^j \mid j \geq i\}$, respectively.

Lemma 3.2. *Let r be a star-free regular expression. Then $r \cap \sigma_1^* \cdots \sigma_n^*$ is equivalent to a disjunction ρ_r*

of expression of the form $\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$ *where each* $*_j \in \{=, \geq\}$ *and* $i_j \in \mathbb{N}$. *Moreover,* $i_1, \ldots, i_n \leq |r|$, *the size of* ρ_r *is exponential in* $|r| + n$, *and* ρ_r *can be computed in time exponential in* $|r| + n$.

Note that R does *not* typecheck w.r.t. d iff

- there is a path $v_1, \ldots, v_k$ in R where (i) v_1 is a child of the root; (ii) $\text{lab}(v_i) = (\sigma_i, \varphi_i(\bar{x}_1, \ldots, \bar{x}_i))$, for $i \in \{1, \ldots, k\}$; (iii) v_k has precisely n children with labels $(\delta_1, \psi_1(\bar{x}, \bar{y}_1)), \ldots, (\delta_n, \psi_n(\bar{x}, \bar{y}_n))$ and in that order; and
- there is an $\mathcal{A}$ with elements $\bar{a} := \bar{a}_1, \ldots, \bar{a}_k$ such that (i) $\mathcal{A} \models \Phi$; (ii) $\mathcal{A} \models \varphi_i(\bar{a}_1, \ldots, \bar{a}_i)$ for each $i = 1, \ldots, k$; and (iii) $\delta_1^{i_1} \cdots \delta_n^{i_n} \notin d(\sigma_k)$ with $|\{\bar{b} \mid \mathcal{A} \models \psi_j(\bar{a}, \bar{b})\}| = i_j$ for all $j = 1, \ldots, n$.

Let $d(\sigma_k)$ be represented by the star-free regular expression r. So, $\delta_1^{i_1} \cdots \delta_n^{i_n} \notin L(r)$. Since for each $\mathcal{A}$, this string will be of the form $\delta_1^* \cdots \delta_n^*$, it suffices to restrict attention to $\neg r \cap \delta_1^* \cdots \delta_n^*$. By Lemma 3.2, $\neg r \cap \delta_1^* \cdots \delta_n^*$ is equivalent to a disjunction, of exponential size, of expressions of the form $\delta_1^{*_1 j_1} \cdots \delta_n^{*_n j_n}$ where each $*_i \in \{=, \geq\}$ and $j_i \leq |r|$. Let D be a particular disjunct $\delta_1^{*_1 j_1} \cdots \delta_n^{*_n j_n}$ such that there is a structure $\mathcal{A}$ with elements $\bar{a} := \bar{a}_1, \ldots, \bar{a}_k$ with

(1) $\mathcal{A} \models \Phi$ and $\mathcal{A} \models \varphi_i(\bar{a}_1, \ldots, \bar{a}_i)$ for each i; and

(2) $|\{\bar{b} \mid \mathcal{A} \models \psi_i(\bar{a}, \bar{b})\}| *_i j_i$ for $i = 1, \ldots, n$.

We next show there is a structure $\mathcal{B}$ of size polynomial in $|R| + |d| + |\Phi|$ satisfying (1) and (2). To see this, we introduce some notation. Suppose $\Phi = \bigcup_\ell \exists \bar{x}_\ell^\alpha \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$, $\varphi_i(x_1, \ldots, x_i) = \exists \bar{x}_i^\varphi \gamma_i(x_1, \ldots, x_i, \bar{x}_i^\varphi)$, for each $i = 1, \ldots, k$, and $\psi_i(\bar{x}, \bar{y}_i) = \exists \bar{x}_i^\psi \beta_i(\bar{x}, \bar{y}_i, \bar{x}_i^\psi)$, for each $i = 1, \ldots, n$.

For each ℓ, pick a tuple $\bar{a}_\ell^\alpha$ such that $\mathcal{A} \models \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{a}_\ell^\alpha, \bar{y}_\ell^\alpha)$. Let E_1 be the set of these elements. Next, pick $a_1, \ldots, a_n$ and for each $i \in \{1, \ldots, k\}$ pick a tuple $\bar{a}_i^\varphi$ such that $\mathcal{A} \models \gamma_i(a_1, \ldots, a_i, \bar{a}_i^\varphi)$. Let E_2 be the set of these elements. Further, for $i = 1, \ldots, n$, pick j_i tuples $\bar{b}_i$ and for each such tuple pick a tuple $\bar{a}_i^\psi$ such that $\mathcal{A} \models \beta_i(a_1, \ldots, a_i, \bar{b}_i, \bar{a}_i^\psi)$. Let E_3 be the set of these elements. Note that the size of $E := E_1 \cup E_2 \cup E_3$ is at most polynomial in $|R| + |d| + |\Phi|$. Clearly, $|\{\bar{b} \mid \mathcal{A}_{|E} \models \psi_j(\bar{a}, \bar{b})\}| *_i j_i$ for $i = 1, \ldots, n$. Moreover, $\mathcal{A}_{|E} \models \Phi$. The latter follows by a standard argument (see, e.g., [6]). Indeed, for each ℓ, $(\mathcal{A}, E_1) \models \forall \bar{y}_\ell^\alpha \alpha_j(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$, where the elements in E_1 are taken as constants. As these resulting sentences are universal, $(\mathcal{A}_{|E}, E_1) \models \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$ for each ℓ. Hence, $\mathcal{A}_{|E} \models \exists \bar{x}_\ell^\alpha \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$ for each ℓ. Then take $\mathcal{B}$ as $\mathcal{A}_{|E}$.

Hence, to look for a database that satisfies the disjunct D it suffices to guess one of exponential size. Recall that if we find such an $\mathcal{A}$, R does *not* typecheck w.r.t. d. The overall algorithm consists of two stages: (i) For every node v labeled with σ and with children $(\delta_1, \psi_1(\bar{x}, \bar{y}_1)), \ldots, (\delta_n, \psi_n(\bar{x}, \bar{y}_n))$, compute the normal form for $\neg d(\sigma) \cap \delta_1^* \cdots \delta_n^*$ as specified in Lemma 3.2. There is a linear number of nodes, so altogether we need exponential time. (ii) Subsequently, guess a path $v_1, \ldots, v_k$, a disjunct D, and a structure $\mathcal{A}$ such that the above holds. As described above this can all be done in NEXPTIME. □

The following result shows that decidability of typechecking holds even when DTDs use full regular languages, as long as the conjunctive queries in the TreeQL program are restricted to be projection-free and the constraints are FDs. The proof is non-trivial and is based on Ramsey's theorem. It is similar to the proof of an analogous but harder result in [3]. A self-contained proof will be provided in the full paper.

Theorem 3.3. TC[projection-free $\text{CQ}^{=,\neg}$, REG, FD] *is decidable.*

It remains open whether the projection-free restriction can be removed or whether the class of constraints can be extended.

4 Complexity

Theorem 3.1 provides an upper bound of CONEXPTIME on the complexity of type-checking. We show in this section that this is tight. Our proof requires negation and inequality in CQs. However, we show that even without these, typechecking remains intractable, more precisely DP-hard.[2] Nevertheless, by further restricting the structure of CQs and $\mathcal{SL}$-formulas we obtain a PTIME algorithm for typechecking. To this end define $\mathcal{SL}^r$ as the fragment of $\mathcal{SL}$ where there are no occurrences of the form $\sigma^{=i}$ and all occurrences of the form $\sigma^{\geq i}$ are such that $i \in \{0, 1\}$. We abbreviate $\sigma^{\geq 1}$ simply by σ. This fragment already suffices to obtain the next lower bound.

Theorem 4.1. TC[$\text{CQ}^{\neg,=}$, $\mathcal{SL}^r$, $\emptyset$] *is hard for* CONEXPTIME.

Proof. The proof consists of a reduction from the satisfiability problem of FO($\exists^*\forall^*$) sentences without equality, which is known to be hard for NEXPTIME (see, e.g., [6]), to the complement of the typechecking problem.

Let φ be a formula of the form $\exists x_1, \ldots, x_n \forall y_1, \ldots, y_m \psi(\bar{x}, \bar{y})$ over the relations $R_1, \ldots, R_k$ without equality. The input database

[2] Recall that DP properties are of the form $\sigma_1 \wedge \sigma_2$ where $\sigma_1 \in$ NP and $\sigma_2 \in$ CO-NP.

for the TreeQL program consists of the relations $D_1, \ldots, D_n, R_1, \ldots, R_k$. The sets $D_1, \ldots, D_n$ will be singletons and will serve as the interpretations for the variables $x_1, \ldots, x_n$.

We have to check whether there is a database $\mathcal{A}$ with a tuple $\bar{d}$ such that $\mathcal{A} \models \forall \bar{y} \psi(\bar{d}, \bar{y})$. We test the converse, that is $\mathcal{A} \not\models \forall \bar{y} \psi(\bar{d}, \bar{y})$ or equivalently $\mathcal{A} \models \exists \bar{y} \neg \psi(\bar{d}, \bar{y})$. Assume that $\neg\psi$ is of the form $\bigvee_{j=1}^{k} L_j(\bar{x}, \bar{y})$ where each $L_j(\bar{x}, \bar{y})$ is a conjunction $\bigwedge C$ of atomic formulas and negations thereof. Thus, each L_j is a projection-free query in $\mathrm{CQ}^{\neg}$. We define a TreeQL program as follows: the root is labeled with 'result' and has exactly one child labeled with

$$(D, \bigwedge_{i=1}^{n} D_i(x_i))$$

giving the required interpretation to the x_is. Further, D has the following children

1. for each $i = 1, \ldots, n$, $(\mathrm{two}_i, \exists z_i \exists z_i'(D_i(z_i) \wedge D_i(z_i') \wedge z_i \neq z_i'))$, indicating that D_i has at least two elements; and
2. for each $j = 1, \ldots, k$, $(@_j, L_j(\bar{x}, \bar{y}))$.

The output DTD d is of the following form $d(\mathrm{result}) := \mathrm{true}$ and

$$d(D) := \bigvee_{i=1}^{n} \mathrm{two}_i \vee \bigvee_{j=1}^{k} @_j.$$

Suppose the TreeQL program R does not typecheck. Then at least one D and none of the two_is appear. That is, all D_i are singleton sets. Let $D_i = \{d_i\}$ for each i. Further, none of the $@_j$s appear. Hence, $\mathcal{A} \not\models \exists \bar{y} \neg \psi(\bar{d}, \bar{y})$. Hence, $\mathcal{A} \models \exists \bar{x} \forall \bar{y} \psi$ and φ is satisfiable. Conversely, if $\mathcal{A}$ is a model of φ and we instantiate $D_1, \ldots, D_n$ with the witnesses for the existential quantifiers then R does not typecheck for $\mathcal{A} \cup \{D_1, \ldots, D_n\}$. □

Although it is unclear whether in Theorem 4.1, negation or inequality can be dispensed with, we show that in any case the complexity of the problem, even for the standard case, remains intractable. Indeed, one can easily reduce the containment of conjunctive queries and propositional validity to typechecking. $\mathrm{CQ}^{\neq}$ denotes CQ with inequality.

Proposition 4.2. *1.* $\mathrm{TC}[\mathrm{CQ}, \mathcal{SL}^r, \emptyset]$ *is* DP*-hard.*

2. $\mathrm{TC}[\mathrm{CQ}^{\neq}, \mathcal{SL}^r, \emptyset]$ *is* Π_2^p*-hard.*

The proof of Proposition 4.2 implies that, in order to have a PTIME algorithm for typechecking, we must at least restrict the queries so that testing containment is in PTIME and that validity of the $\mathcal{SL}^r$ formulas used must be in PTIME. We present one set of restrictions that leads to a PTIME typechecking test. Let CQ^k denote the conjunctive queries in FO^k, i.e. the set of conjunctive queries using at most k variables. Such queries can be evaluated in combined complexity PTIME [11, 20]. We restrict TreeQL programs as follows: there exists some k such that, for each node v in the program, the conjunction of all queries of nodes along the path from root to v is in CQ^k. Furthermore, no distinct siblings v, v' in the query tree have labels (a, φ) and (a, φ') for the same $a \in \Sigma$. We call such a program *k-bounded* and denote the set of k-bounded TreeQL programs by TreeQL^k. Finally, we also need a restriction on the $\mathcal{SL}^r$ formulas used in the DTD: they are in conjunctive normal form. We call such $\mathcal{SL}^r$ formulas *conjunctive.*

Theorem 4.3. $\mathrm{TC}[CQ^k$, conjunctive $\mathcal{SL}^r, \emptyset]$ *is in* PTIME *for* TreeQL^k *programs.*

Proof. Let R be a TreeQL^k program and let d be a DTD using conjunctive $\mathcal{SL}^r$ formulas. We assume w.l.o.g. that every bound variable occurs only once and is different from any free variable. For every non-leaf node v of R with children $v_1, \ldots, v_n$, we do the following. Let $d(\mathrm{lab}(v)) = \varphi_v$, where $\varphi_v = \wedge_i C_i$ and each C_i is a disjunction of positive or negated a_i's. Further, let γ be the conjunction of the formulas occurring in labels along the path from *root* to v. The program typechecks w.r.t. v if for every input, the sequence of children of v in the output satisfies each of the C_i's. So it is enough to typecheck separately with respect to each of the C_i's. Each C_i is of the form $a_1 \vee \ldots \vee a_k \vee \neg b_1 \vee \ldots \neg b_m$. For each $a \in \Sigma$, let ψ_a denote the formula associated to the unique child of v labeled with a. There are three cases to consider:

1. $k > 0$ and $m > 0$. Then C_i is $(b_1 \wedge \ldots \wedge b_m) \rightarrow (a_1 \vee \ldots \vee a_k)$. We must check that

$$\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma) \rightarrow \exists((\psi_{a_1} \wedge \gamma) \vee \ldots \vee (\psi_{a_k} \wedge \gamma))$$

where the $\exists$ quantify all variables on the left, resp. righthand sides. From standard conjunctive query techniques it follows that the above holds iff there exists j such that

$$\exists(\psi_{b_1} \wedge \ldots \psi_{b_m} \wedge \gamma) \rightarrow \exists(\psi_{a_j} \wedge \gamma).$$

This in turn holds iff the result of evaluating the conjunctive query $\exists(\psi_{a_j} \wedge \gamma)$ on the canonical structure associated to the matrix of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ is true. Since $\exists(\psi_{a_j} \wedge \gamma)$ is in CQ^k, this can be checked in PTIME.

2. $m = 0$. This amounts to testing that $\exists((\psi_{a_1} \wedge \gamma) \vee \ldots \vee (\psi_{a_k} \wedge \gamma))$ is true on every input. This is false on the empty input, so the program does not typecheck.

3. $k = 0$. Since $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ is always satisfiable, this never typechecks. □

5 Undecidability Results

We have seen in the previous section that TC[CQ$^{\neg,=}$, SF, FO($\exists^*\forall^*$)] is decidable. This is a fairly tight bound. Indeed, we next show that even minor extensions lead to undecidability. We consider several extensions of the output DTDs, TreeQL queries, and integrity constraints. Specifically, we consider (*i*) specialization, (*ii*) virtual nodes, and (*iii*) acyclic inclusion dependencies (AcID), and show that typechecking becomes undecidable with each of these extensions. Another parameter in the formalism is the class of string languages used by DTDs. Recall that decidability still holds if we replace SF by REG when restricting to projection-free CQs and omit integrity constraints. We show that this most likely cannot be extended beyond REG: allowing *deterministic* CFLs (DCFL) in DTDs leads to undecidability.

We first consider the impact of augmenting DTDs with specialization.

Theorem 5.1. TC[projection-free CQ, $\mathcal{SL}^r_{\text{spec}}$, $\emptyset$] *is undecidable.*

Proof. We use a reduction from satisfiability of first-order logic formulas over graphs without equality, which is well known to be undecidable (see, e.g., [6]). The satisfiability problem is to check, given an FO formula ψ, whether there is a non-empty graph $\mathcal{A}$ such that $\mathcal{A} \models \psi$. Let φ be the negation of ψ. We give the reduction by example. Assume $\varphi = \exists x_1 \forall x_2 \exists x_3 \delta(x_1, x_2, x_3)$, where δ is quantifier-free and in disjunctive normal form, that is, of the form $\bigvee_{i=1}^{m} L_i$, where each L_i is of the form $P^i \wedge \bigwedge_{j=1}^{m_i} N^i_j$ where P^i is a conjunction of atomic formulas and each N^i_j is the negation of a single atomic formula. For a negated atomic formula N we denote the unnegated formula by $\tilde{N}$. Recall that atomic formulas can only be of the form $E(x_i, x_j)$.

Consider the TreeQL(CQ) program R depicted in Figure 1. By L_i we denote the sequence

$$(P^i, P^i)(N^i_1, \tilde{N}^i_1) \ldots (N^i_{m_i}, \tilde{N}^i_{m_i}).$$

Recall that the first component of the pair is a label while the second one is a formula. Intuitively, every occurrence of an X_i in the output tree represents a value assignment for the variable x_i. The specialized DTD then takes care of the quantification pattern of φ. Indeed, it should verify that there is an X_1-node such that for all its X_2-children there is an X_3-node that satisfies δ. To this end let $\Sigma' = \{Y_i, X_i \mid i \in \{1, \ldots, n\}\} \cup \{\text{result}\}$. Intuitively, whenever a node is labeled Y_i, this indicates that the path from the root to this node can be extended to a satisfiable path. Define $d(\text{result}) := Y_1 \vee \varepsilon$, $d(Y_1) := Y_2 \wedge \neg X_2$, $d(Y_2) := Y_3$, and $d(X_1) := d(X_2) := d(X_3)$. Here, ε makes sure the empty graph typechecks. Finally, set for each i, $\mu(X_i) := X_i$ and $\mu(Y_i) := X_i$. Clearly, R typechecks w.r.t. d iff $\mathcal{A} \models \varphi$ for every non-empty structure $\mathcal{A}$.

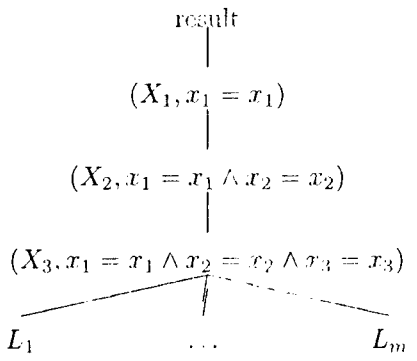


Figure 1: The TreeQL program R.

One can get rid of equality in the CQ's by introducing a relation containing all elements in the active domain. Details omitted. □

The next result shows that typechecking becomes undecidable when queries can use virtual nodes. The proof is similar to the proof of Theorem 5.1 and is omitted.

Theorem 5.2. TC[projection-free CQ$_{virt}$, SF, $\emptyset$] *is undecidable.*

Remark 5.3. The undecidability result in Theorem 5.5 requires DTDs using SF formulas. The next proposition shows that restricting the DTD language to $\mathcal{SL}$ renders typechecking decidable, even when virtual nodes are allowed.

Proposition 5.4. TC[CQ$^{\neg,=}_{virt}$, $\mathcal{SL}$, FO($\exists^*\forall^*$)] *is decidable.* □

Next, we consider the effect of the constraints on decidability. We show that even the usually well-behaved unary AcIDs (which are not definable in FO($\exists^*\forall^*$)) render typechecking undecidable.

Theorem 5.5. TC[CQ$^{\neg,=}$, $\mathcal{SL}^r$, unary AcIDs] *is undecidable.*

Proof. We consider the fragment of FO consisting of formulas of the form $\forall x\varphi(x)$ where φ is a quantifier-free formula over the vocabulary of two unary functions f and g. It is well-known that it is undecidable whether there is a non-empty structure $\mathcal{A}$ such that $\mathcal{A} \models \forall x\varphi(x)$ (see e.g. [6]). The schema of the input database consists of the two binary relations F and G (representing the functions f and g), and a unary relation D representing the active domain of the structure. Using D will allow to get rid of circular dependencies.

First, we have to make sure that F and G are indeed functions, that their domain is D, and their range is included in D. These are specified by the *cyclic* unary inclusion dependencies

$$\begin{array}{llll} (a) & F[1] \subseteq D[1] & (e) & D[1] \subseteq F[1] \\ (b) & G[1] \subseteq D[1] & (f) & D[1] \subseteq G[1] \\ (c) & F[2] \subseteq D[1] & & \\ (d) & G[2] \subseteq D[1]. & & \end{array}$$

However, we will only keep the dependencies (e) and (f): we show that (a)–(d) can be expressed by the TreeQL program itself. We next describe this TreeQL program in detail. We first check whether the inclusion dependency (a) holds. If not we generate the flag (a)_does_not_hold.

result

$$((a)\texttt{_does_not_hold}, \exists x \exists y(F(x,y) \wedge \neg D(x))).$$

The same is done for the dependencies (b)–(d). Next we have to check whether F is indeed a function and not a relation. For instance, both (a,b) and (a,c), with $b \neq c$, could belong to F. This can be detected as follows

result

$$(\texttt{wrong_}F, \exists x \exists y \exists z(F(x,y), F(x,z) \wedge y \neq z)).$$

The same is done for G. In particular, if G is a relation and not a function then the flag wrong_G is raised.

We test whether $\mathcal{A} \not\models \forall x\varphi(x)$, that is, $\mathcal{A} \models \exists x\neg\varphi(x)$. We can rewrite $\exists x\neg\varphi(x)$ to

$$\bigvee_{i=1}^{n} (\exists x) L_i,$$

where each L_i is of the form $\bigwedge_{j=1}^{m_i} C_j^i$ where each C_j^i is an equality or an inequality between terms. For instance, $C_1 \equiv fgx = ffx$ (parenthesis omitted for clarity) or $C_2 \equiv fgx \neq ffx$. Obviously, there is a canonical way to associate a $\text{CQ}^{=,\neg}$ with each C. For instance,

$$\varphi_{C_1}(x) = \exists y_2, y_3, z_2, z_3(G(x,y_2) \wedge F(y_2,y_3) \wedge F(x,z_2) \wedge F(z_2,z_3) \wedge y_3 = z_3),$$

and

$$\varphi_{C_2}(x) = \exists y_2, y_3, z_2, z_3(G(x,y_2) \wedge F(y_2,y_3) \wedge F(x,z_2) \wedge F(z_2,z_3) \wedge y_3 \neq z_3).$$

Further, we define φ_{L_i} as $\varphi_{C_1^i}(x) \wedge \ldots \wedge \varphi_{C_{m_i}^i}(x)$. The just described part of the TreeQL query is then of the form:

result

$$(L_1, \exists x\varphi_{L_1}(x)) \quad \ldots \quad (L_n, \exists x\varphi_{L_n}(x)).$$

Hence, $\mathcal{A} \not\models \forall x\varphi(x)$ whenever one of the error flags L_i is raised.

Finally, we have to make sure that D is non-empty. Therefore we have

result

$$(D\text{-not_empty}, \exists z D(z)).$$

The final TreeQL program is the concatenation of the previous programs (that is, the concatenation of all children under one result node). Note that a non-empty input structure for which $\mathcal{A} \models \forall x\varphi(x)$ simply generates the tree result(D-not_empty). The output DTD d then maps result to D-not_empty $\rightarrow$ error, where error is the disjunction over all error flags. If R does not typecheck w.r.t. d, then there is an $\mathcal{A}$ and an ordering $<$ such that $R(\mathcal{A},<) \notin L(d)$. By construction, $\mathcal{A}$ is non-empty and no error flag is raised. Therefore, $\mathcal{A}_{|D} \models (\forall x)\varphi(x)$. Conversely, if there is an $\mathcal{A}$ such that $\mathcal{A} \models \forall x\varphi(x)$ then for every ordering $<$, $R(\mathcal{A} \cup D,<) \notin L(d)$, where D is interpreted by the active domain of $\mathcal{A}$. □

Theorem 3.3 showed that typechecking remains decidable even for DTDs using full regular languages, as long as the queries are restricted to be projection free. As shown next, going beyond regular languages quickly leads to undecidability.

Theorem 5.6. TC[projection-free CQ, DCFL, $\emptyset$] *is undecidable.*

Proof. The proof is a reduction from Hilbert's tenth problem, diophantine equations, well-known to be undecidable [12]. We consider the following variant.

For a polynomial $P(x_1, \ldots, x_n)$ with integer coefficients, are there positive integers $i_1, \ldots, i_n$ such that $P(i_1, \ldots, i_n) = 0$? We only give the reduction by example. The general case is a straightforward generalization. Consider, for instance, the polynomial $2xy - x^2 + 1$. The input database consists of two sets X and Y where the cardinalities of X and Y stand for the numbers x and y, respectively. We describe a TreeQL program that generates from X and Y sequences of a's and b's. A positive term in P generates a's while a negative one generates b's. Hence, an a stands for $+1$, and a b stands for -1. The output DTD states that the number of a's differs from the number of b's. This holds iff $|X|$ and $|Y|$ do not form a solution to P, and the language specified by the DTD can easily be recognized by a deterministic PDA. The TreeQL program is a tree of depth one. For the example polynomial, the nodes under the root are:

$$(a, X(x) \wedge Y(y)) \cdot (a, X(x) \wedge Y(y)) \cdot (b, X(x_1) \wedge X(x_2)) \cdot (a, \text{true}).$$

Here, the first two symbols correspond to the term $2xy$ and generate a's as the term is positive; similarly, the third and the fourth symbol correspond to $-x^2$ and $+1$, respectively. The output generates sequences of a's and b's. The deterministic PDA accepts when the number of a's is different from the number of b's. Hence, the TreeQL program typechecks iff the diophantine equation has no positive solution. □

6 Conclusions

We investigated the problem of typechecking XML views of relational databases satisfying given integrity constraints. This is a practically important problem in the context of the Web, where relational databases must be exported in XML form that satisfies target DTDs. The formal query language TreeQL maps first-order relational structures to tree data, and is a faithful abstraction of the view definition language used in the SilkRoute prototype. The results of the paper trace a fairly tight border of decidability for the typechecking problem. The parameters considered include features of the query language, of the DTDs, and the class of integrity constraints satisfied by the relational database. The proofs bring into play a variety of techniques at the confluence of finite-model theory, language theory, and combinatorics.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. To apper in *PODS* 2001.

[4] D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. XML schema part 1: Structures, May 1999. `http://www.w3.org/TR/xmlschema-1/`.

[5] P. Biron and A. Malhotra. XML schema part 2: Datatypes, May 1999. `http://www.w3.org/TR/xmlschema-2/`.

[6] E. Börger, E. Grädel, and Y. Gurevich. *The classical decision problem.* Springer, 1997.

[7] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998.

[8] E. F. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13 (6), pp. 377-387, 1970.

[9] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1995.

[10] M. Fernandez, D. Suciu and W. Tan. SilkRoute: trading between relations and XML. *Proceedings of the WWW9 Conference*, Amsterdam, pp. 723–746, 2000,

[11] N. Immerman. Upper and lower bounds for first-order expressibility. *J. of Computer and System Sciences*, vol.25, pp. 76–98, 1982.

[12] Yuri V. Matiyasevich. *Hilbert's tenth problem.* Foundations of Computing Series. MIT Press, 1993.

[13] R. McNaughton and S. Papert. *Counter-Free Automata.* MIT Press, 1971.

[14] T.Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 11-22, 2000.

[15] John C. Mitchell. *Foundations for Programmng Languages*, MIT Press, 1996.

[16] F. Neven and T. Schwentick. Unordered DTDs. Unpublished manuscript, 1999.

[17] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 35-46, 2000.

[18] W. Thomas. Languages, automata, and logic. In Rozenberg and Salomaa, *Handbook of Formal Languages*, volume III, chapter 7. Springer, 1997.

[19] R. van der Meyden. The complexity of querying infinite data about linearly ordered domains *JCSS*, 54(1):113-135, 1997.

[20] M. Vardi. On the complexity of bounded-variable queries. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 266–276, 1995.

A Model-Theoretic Approach to Regular String Relations*

Michael Benedikt† Bell Labs
Leonid Libkin§ U. Toronto
Thomas Schwentick¶ U. Jena
Luc Segoufin‖ INRIA

Abstract

We study algebras of definable string relations – classes of regular n-ary relations that arise as the definable sets within a model whose carrier is the set of all strings. We show that the largest such algebra – the collection of regular relations – has some quite undesirable computational and model-theoretic properties. In contrast, we exhibit several definable relation algebras that have much tamer behavior: for example, they admit quantifier elimination, and have finite VC dimension. We show that the properties of a definable relation algebra are not at all determined by the one-dimensional definable sets. We give models whose definable sets are all star-free, but whose binary relations are quite complex, as well as models whose definable sets include all regular sets, but which are much more restricted and tractable than the full algebra of regular relations.

1 Introduction

In the past 40 years, various connections between logic, formal languages and automata have been explored in great detail. The standard setting for connecting logical definability with various properties of formal languages is to represent strings over a finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$ as first-order structures in the signature $(P_{a_1}, \ldots, P_{a_n}, <)$, so that the structure M_s for a string s of length k has the universe $\{1, \ldots, k\}$, with $<$ being the usual ordering, and P_{a_i} being the set of the positions l such that the lth character in s is a_i. Then a sentence Φ of some logic $\mathcal{L}$ defines a language $L(\Phi) = \{s \in \Sigma^* \mid M_s \models \Phi\}$. Two classical results on logic and language theory state that languages thus definable in monadic second-order logic (MSO) are precisely the regular languages [8], and the languages definable in first-order logic (FO) are precisely the star-free languages [25]. For a survey, see [28, 29].

An alternative approach to definability of strings, based on classical infinite model theory rather than finite model theory, dates back to [8, 10]. One considers an infinite structure M consisting of $\langle \Sigma^*, \Omega \rangle$, where Ω is a set of functions, predicates and constants on Σ^*. One can then look at definable sets, those of the form $\{\vec{a} \mid M \models \varphi(\vec{a})\}$, where φ is a first-order formula in the language of M. A well-known result links definability with traditional formal language theory. Let Ω_{reg} consist of unary functions l_a, $a \in \Sigma$, binary predicates $\text{el}(x, y)$ and $x \preceq y$, where $l_a(x) = x \cdot a$, $\text{el}(x, y)$ states that x and y have the same length, and $x \preceq y$ states that x is a prefix of y. Let $\mathbf{S}_{\text{len}}$ be the model $\langle \Sigma^*, \Omega_{\text{reg}} \rangle$ (we will explain the notation later). Then subsets of Σ^* definable in $\mathbf{S}_{\text{len}}$ are precisely the regular languages [8, 10, 9].

An advantage of the "model-theoretic approach" is that one immediately gets an extension of the notion of recognizability from string languages to n-ary string relations for arbitrary n. One gets an algebra of n-ary string relations for every n, and these algebras automatically have closure under projection and product, in addition to the boolean operations. In the case of the model $\mathbf{S}_{\text{len}}$ above, this algebra is not new: in fact, the definable n-ary relations are exactly the ones recognizable under a natural notion of automaton running over n-tuples [10, 15].

An obvious question to ask, then, is whether *new* algebras of string relations arise through the model-theoretic approach. In particular, if we restrict the signature Ω to be less expressive than Ω_{reg}, do we get new relation algebras lying within the recognizable re-

*Part of this work was done while the second and the third authors visited INRIA, and the second and the fourth authors visited Mainz.

†Bell Laboratories, 263 Shuman Blvd, Naperville, IL 60566, USA. E-mail: benedikt@research.bell-labs.com.

§Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu. Research affiliation: Bell Labs.

¶Current address: Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 3, 07740 Jena, Germany. Email: tick@informatik.uni-jena.de. Work done while at U. Mainz.

‖INRIA-Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. E-mail: Luc.Segoufin@inria.fr.

0-7695-1281-X/01 $10.00 © 2001 IEEE

lations?

A natural starting point would be to find a signature that captures properties of the star-free sets. Here again, a simple example leaps out: consider the signature $\Omega_{\text{sf}} = (\preceq, (l_a)_{a\in\Sigma})$, and let $\mathbf{S} = \langle \Sigma^*, \Omega_{\text{sf}} \rangle$. One can easily show that the definable subsets of Σ^* in $\mathbf{S}$ are exactly the star-free ones. Furthermore, we will show that the definable n-ary relations of this model are exactly those definable by regular prefix automata (cf. [1]) whose underlying string automata are counter-free.

Just as there is a significant difference between the complexity-theoretic behavior of regular languages and star-free languages, we find that the model $\mathbf{S}$ is much more tractable, in terms of its model-theory and its complexity than $\mathbf{S}_{\text{len}}$. In particular, we show that $\mathbf{S}$ has quantifier-elimination in a natural relational extension, while $\mathbf{S}_{\text{len}}$ does not.

It would be tempting to think of $\mathbf{S}$ and $\mathbf{S}_{\text{len}}$ as canonical extensions of the notions of regularity and star-free to n-ary relations. However, we will show that in fact there are *many* choices for Ω that share the same one-dimensional definable sets (either star-free or regular). Furthermore, algebras of definable sets may be identical in terms of the string languages they define, but differ considerably in the n-ary string relations in the definable algebra. We thus say that an algebra of definable sets based on $\langle \Sigma^*, \Omega \rangle$, with $\Omega \subseteq \Omega_{\text{reg}}$ is a *regular algebra of definable sets* if the subsets of Σ^* in it (i.e the one-dimensional definable sets of $\langle \Sigma^*, \Omega \rangle$) are exactly the regular sets. We likewise say that the algebra based on definable sets for $\langle \Sigma^*, \Omega \rangle$ is a *star-free algebra of definable sets* if the subsets of Σ^* in the algebra are exactly the star-free sets.

The rest of the paper studies new examples of regular and star-free definable algebras. We give an example of a star-free algebra with considerably more expressive power than the basic star-free algebra $\mathbf{S}$. This model, which we denote by $\mathbf{S}_{\text{left}}$ (as it allows one to add characters on the *left* of a string), shares most of the desirable properties of $\mathbf{S}$: in particular, it has quantifier-elimination in a natural language, and membership test in this algebra has low complexity.

More surprisingly, perhaps, we give examples of regular algebras (which we denote $\mathbf{S}_{\text{reg}}$ and $\mathbf{S}_{\text{reg,left}}$) that are strictly contained in $\mathbf{S}_{\text{len}} = \langle \Sigma^*, \Omega_{\text{reg}} \rangle$. Although the one-dimensional sets in these algebras are still the regular sets, the algebra as a whole shares many of the attractive properties of the star-free languages. In particular, we give quantifier-elimination results for these algebras.

One key motivation for our work comes from the field of databases, in particular, the study of query languages with interpreted operations [3, 5, 19], and constraint databases [23]. In those settings, quantifier-elimination gives one closed-form evaluation for queries; it says that one can evaluate queries whose input is a quantifier-free definable set and get a closed form solution as another quantifier-free definable set. This approach has generally been applied to numerical domains over the reals, since there are several powerful quantifier-elimination results available there. It is natural to extend this approach to databases over strings: the string datatype, after all, is ubiquitous in database applications, and languages such as SQL already give some capability of manipulating star-free sets (via the LIKE predicate) defined from the input data within queries. But in order to extend the constraint-database approach to the string context, we are first required to find definable algebras that admit quantifier-elimination in some natural yet powerful language. (Some of the previous results in this direction considered query languages over undecidable structures [20], or decidable ones but not capable of expressing some very basic operations on strings [14].) The quantifier-elimination results here thus yield new examples where the constraint approach can be applied. In fact, the results we present here were used in [7] to give expressiveness and complexity bounds for the database query languages that arise from several algebras of definable sets.

Our approach was also motivated by the study of *automatic structures* [22, 9], which are a subclass of recursive structures [21], and were introduced recently as a generalization of automatic groups [16]. In an automatic structure $M = \langle \Sigma^*, \Omega \rangle$, every predicate in Ω is definable by a finite automaton. More precisely, an n-ary predicate P is given by a letter-to-letter n-automaton [15, 18]. Such an automaton is a usual DFA whose alphabet is $(\Sigma \cup \{\#\})^n$, $\# \notin \Sigma$. An n-tuple of strings $s_1, \ldots, s_n$ can be viewed as a word of length $\max_i |s_i|$ over the alphabet $\Sigma \cup \{\#\}$, where the jth letter is the tuple $(s_1^j, \ldots, s_n^j)$; here s_k^j is the jth letter of s_k, if $|s_k| \leq j$, and $\#$ otherwise. We then say that a predicate $P \subseteq (\Sigma^*)^n$ is definable by a letter-to-letter n-automaton A if $(s_1, \ldots, s_n) \in P$ iff A accepts $s_1, \ldots, s_n$.

It is known [10, 9] that a structure is automatic iff it can be interpreted in the structure $\mathbf{S}_{\text{len}}$; hence $\mathbf{S}_{\text{len}}$ is in some sense the universal automatic structure. It is interesting then to look at subclasses of automatic structures definable within $\mathbf{S}_{\text{len}}$ that are significantly more restrictive, and that might have stronger model-theoretic or computational properties than a rich structure like $\mathbf{S}_{\text{len}}$. One dividing line we focus on is be-

tween automatic structures that do admit quantifier-elimination in a natural relational language, and those that do not.

Our first result gives a partial answer to open question 0 in [26], which asks whether $\mathbf{S}_{\text{len}}$ itself has quantifier-elimination in a reasonable signature. We show that it does not have quantifier-elimination in any relational signature of bounded arity but does have quantifier-elimination in a signature containing binary functions. The other structures that we study — $\mathbf{S}$, $\mathbf{S}_{\text{reg}}$, $\mathbf{S}_{\text{left}}$ and $\mathbf{S}_{\text{reg,left}}$ — do admit such a quantifier-elimination. A second dichotomy is between automatic structures that admit star-free definable algebras versus those that have regular algebras. We show that the models $\mathbf{S}$ and $\mathbf{S}_{\text{left}}$ have star-free definable algebras, while the model $\mathbf{S}_{\text{reg}}$ does not. Our results indicate that the class of automatic structures that admit star-free definable algebras is richer than one might have guessed.

Organization Section 2 introduces the notation. Section 3 explores the motivating example, the model $\mathbf{S}_{\text{len}}$, and proves a set of results concerning its limitations. In Section 4 we turn to the minimal example of a star-free algebra, the model $\mathbf{S}$, and prove a quantifier-elimination result for this model that contrasts with the negative result proved for $\mathbf{S}_{\text{len}}$. Section 5 extends the results of the previous section to a more complex example of a star-free algebra, the model $\mathbf{S}_{\text{left}}$. Section 6 gives a restriction of $\mathbf{S}_{\text{len}}$ that admits a regular algebra, and proves a quantifier elimination result for this model. The section also connects this model to the minimal model $\mathbf{S}$. Section 7 gives an additional example of a regular algebra, which contains each of the previous examples. Section 8 gives conclusions. All proofs are in the full version [6].

2 Notations

Throughout the paper, Σ denotes a finite alphabet, and Σ^* the set of all finite strings over Σ. We consider a number of operations on Σ^*:

- $x \cdot y$ – concatenation of two strings x and y.
- $x \preceq y$ – x is a prefix of y.
- $l_a(x)$, $a \in \Sigma$, is $x \cdot a$ (adds *l*ast character).
- $f_a(x)$, $a \in \Sigma$, is $a \cdot x$ (adds *f*irst character).
- $|x|$ is the length of string x.
- $x \sqcap y$ is the longest common prefix of the strings x and y.
- $x - y$ – the string z such that $y \cdot z = x$, if it exists, and ϵ otherwise.

Note that $|x|$ does not return a string, so it is not an operation of Σ^*. Instead, we often consider the predicate $\text{el}(x, y)$ which is true iff $|x| = |y|$.

We shall consider several structures on Σ^*. The basic one is the structure $\mathbf{S} = \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma} \rangle$. We could equivalently use unary predicates L_a, where $L_a(x)$ is true for strings of the form $x' \cdot a$. Note that in the presence of $\preceq$, l_a and L_a are interdefinable, and we thus shall use both of them.

We further consider a number of extensions of $\mathbf{S}$. In one of them characters can be added on the *left* as well as on the right. This structure is denoted by $\mathbf{S}_{\text{left}} \stackrel{\text{def}}{=} \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma}, (f_a)_{a\in\Sigma} \rangle$. Another extension, denoted by $\mathbf{S}_{\text{len}}$, adds *length* comparisons via the el predicate (note that using $\preceq$ and el one can express various relationships between lengths of strings, e.g. $|x|\{=, \neq, <, >\}|y|$, $|x| = |y| + k$ for a constant k, etc.). To summarize, we mainly deal with the following structures:

- $\mathbf{S} = \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma} \rangle$;
- $\mathbf{S}_{\text{left}} = \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma}, (f_a)_{a\in\Sigma} \rangle$;
- $\mathbf{S}_{\text{len}} = \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma}, \text{el} \rangle$.

Once we consider regular algebras, we introduce two more structures; however, operations in them will be motivated by quantifier-elimination results for $\mathbf{S}$ and $\mathbf{S}_{\text{left}}$ and thus those structures will be defined later.

There is a very close connection between $\mathbf{S}_{\text{len}}$ and an extension of Presburger arithmetic. Assume that $\Sigma = \{0, 1\}$. Let $\text{val}(n)$, for $n \in \mathbb{N}$, be n in binary, considered as a string in Σ^*. Let $V_2(n)$ be the largest power of 2 that divides n. Then $P \subseteq \mathbb{N}^k$ is definable in $\langle \mathbb{N}, +, V_2 \rangle$ iff $\{(\text{val}(n_1), \ldots, \text{val}(n_k)) \mid (n_1, \ldots, n_k) \in P\}$ is definable in $\mathbf{S}_{\text{len}}$ [8, 10].

Model theory background Let Ω be a finite or countably infinite first-order signature, and M a model over Ω. By $\text{FO}(M)$ we denote the set of all first-order formulae in the language of Ω. The (complete) theory of M, $\text{Th}(M)$, is the set of all sentences in $\text{FO}(M)$ true in M. Two models M and M' over Ω are elementary equivalent if $\text{Th}(M) = \text{Th}(M')$.

We say that M admits *quantifier elimination (QE)* if for every formula $\varphi(\vec{x})$ in $\text{FO}(M)$ there is a quantifier-free formula $\varphi'(\vec{x})$ such that $\forall \vec{x}\ \varphi(\vec{x}) \leftrightarrow \varphi'(\vec{x})$ is true in M.

For a tuple $\vec{a}$ and a model M over Ω, we let $tp_M(\vec{a})$ be the *type* of $\vec{a}$ in M (the set of all formulae of $\text{FO}(M)$

satisfied by $\vec{a}$), and $atp_M(\vec{a})$ be the atomic type in M (the set of all quantifier-free formulae of FO(M) satisfied by $\vec{a}$). If A is a subset of M, $tp_M(\vec{a}/A)$ is the type of $\vec{a}$ over A in M (the set of all FO-formulae over $\Omega \cup A$ satisfied by $\vec{a}$).

A *ω-saturated model* M over Ω is a model such that each consistent type over a finite set A in FO(M) is satisfied in M. It is known [11] that every model M over Ω has an elementary equivalent ω-saturated model M^*.

Isolation, VC-dimension Let T be a theory over Ω and M be a model of T. A subset A of M is said to be *pseudo-finite* if $(M, A) \models F(T, P)$, where P is a unary predicate, and $F(T, P)$ is the set of all formulae of FO($\Omega \cup P$) satisfied by all finite sets of elements in any model of T.

If p is a type over A in M, a subset q of p *isolates* p if p is the only type over A in M containing q. A complete theory T over Ω is said to have the *strong isolation property* if for any model M of T and any pseudo-finite set A and any element a in M, there is a finite subset A_0 of A such that $tp_M(a/A_0)$ isolates $tp_M(a/A)$. We say that it has the *isolation property* if a countable A_0 exists as above.

Isolation is an interesting property in the database context because it implies certain collapse results for query languages [3, 17] and it is used for that purpose in [7]. Here we use it to provide bounds on the VC-dimension of definable families.

For a family $\mathcal{C}$ of subsets of a set U, and a set $F \subseteq U$, we say that $\mathcal{C}$ *shatters* F if $\{F \cap C \mid C \in \mathcal{C}\}$ is the powerset of F. The *VC-dimension* of $\mathcal{C}$ is the maximum cardinality of a finite set shattered by $\mathcal{C}$ (or ∞, if arbitrarily large finite sets are shattered by $\mathcal{C}$). This concept is fundamental to learning theory, as finite VC-dimension of a hypothesis space is equivalent to learnability (PAC-learnability) [2, 4].

Now consider a structure $M = \langle \Sigma^*, \Omega \rangle$, and a FO($M$) formula $\varphi(\vec{x}, \vec{y})$. For each $\vec{a}$, let $\varphi(\vec{a}, M) = \{\vec{b} \mid M \models \varphi(\vec{a}, \vec{b})\}$. The family of sets $\varphi(\vec{a}, M)$, where $\vec{a}$ ranges over all tuples over M, is called a *definable family*. We say that M has finite VC-dimension if every definable family has finite VC-dimension. In particular, this implies learnability of concepts defined in FO over M.

3 Regular algebra based on $\mathbf{S}_{\text{len}}$

As mentioned in the introduction, $\mathbf{S}_{\text{len}}$ is the canonical automatic structure, and relations definable in $\mathbf{S}_{\text{len}}$ are precisely the *regular relations*, that is, k-ary definable relations are precisely those given by letter-to-letter k-automata [9, 10]. In particular, this gives a normal form for $\mathbf{S}_{\text{len}}$-formulae. We introduce a new type of *length-bounded quantifiers* of the form $\exists |x| \leq |y|$ and $\forall |x| \leq |y|$. A formula $\exists |x| \leq |y| \varphi$ is meant as an abbreviation for $\exists x(|x| \leq |y|) \wedge \varphi$. Since every finite automaton can be simulated by a length-bounded FO($\mathbf{S}_{\text{len}}$) formula, we conclude that each FO($\mathbf{S}_{\text{len}}$) formula is equivalent to a length-bounded FO($\mathbf{S}_{\text{len}}$) formula. Note that this result can also be shown by a straightforward Ehrenfeucht-Fraïssé game argument.

The universal property of $\mathbf{S}_{\text{len}}$ mentioned above indicates that $\mathbf{S}_{\text{len}}$ may be "too rich" in relations for many applications. We present evidence for this by addressing the open question of [12, 26] whether $\mathbf{S}_{\text{len}}$ has quantifier elimination in a reasonable signature. One first needs to define what "reasonable" means here. Clearly, every structure has quantifier elimination in a sufficiently large expansion of the signature: add symbols for all definable predicates, for example. One can thus take reasonable to mean a finite expansion, but this is not satisfactory: for example, Presburger arithmetic has quantifier elimination in an infinite signature $(+, <, 0, 1, (\text{mod } k)_{k>1})$. Note however that in this example, the maximum arity of the predicates and functions is 2. In fact, it appears to be a common phenomenon that when one proves quantifier elimination in an infinite signature, there is an upper bound on the arity of functions and predicates in it.

We thus view this condition as necessary for a signature to be "reasonable". In general, a reasonable signature might contain relation symbols as well as function symbols. Nevertheless, we can rule out the possibility of a reasonable, purely relational signature for which $\mathbf{S}_{\text{len}}$ has quantifier elimination. This is in contrast to the weaker structures that we consider, all of which have quantifier elimination in a relational signature of bounded arity. Let $\mathbf{S}_{\text{len}}^{(n,m)}$ be the expansion of $\mathbf{S}_{\text{len}}$ with all definable predicates of arity at most n, and definable functions of arity m. We show the following:

Theorem 1 *(a) For any $n \geq 0$, and $m = 0, 1$, $\mathbf{S}_{\text{len}}^{(n,m)}$ does not have QE. In particular, there is a property definable in $\mathbf{S}_{\text{len}}$ which is not a Boolean combination of at most n-ary definable predicates in $\mathbf{S}_{\text{len}}$.*

(b) $\mathbf{S}_{\text{len}}^{(1,2)}$, the expansion of $\mathbf{S}_{\text{len}}$ with all unary predicates and binary functions, has QE.

Proof sketch. For (a), the property is whether for an N-tuple of strings, for sufficiently large N, there is a position i such that the ith symbol of all N strings is 0. For (b), we show a stronger result, assuming that Σ

contains $\{0,1\}$. We prove QE in a signature that contains the bitwise *and*, *or*, and *not* functions, left and right shifts, and the following two families of functions. $\mathrm{Fil}_\sigma(w)$ has a 1 at position i iff $w[i] = \sigma$ and a 0 otherwise, and $\mathrm{Pat}_{j,k}(w)$ has the same length as w and has a 1 at position i iff $i \bmod k = j$ and a 0 otherwise, where $j < k$.

In cases of both (a) and (b), the proofs are based on automata representations of definable sets, cf. [9]. □

Our next result shows another model-theoretic and computational shortcoming of $\mathbf{S}_{\mathrm{len}}$: namely, a single formula $\varphi(x,y)$ can define a widely varying collection of relations as we let the parameter x vary. We formalize this through the notion of VC-dimension.

Proposition 1 *There are definable families in* $\mathbf{S}_{\mathrm{len}}$ *that have infinite VC-dimension.* □

4 Star-free algebra based on S

We now turn to the most obvious analog of $\mathbf{S}_{\mathrm{len}}$ for the star-free sets. This is the model $\mathbf{S}$, which is the most basic model among those studied in the paper. We show that it has remarkably nice behavior: it admits effective QE in a rather small extension to the signature. This immediately tells us that definable subsets of Σ^* are precisely the star-free languages. We then characterize the n-dimensional definable relations in $\mathbf{S}$ by their closure properties, and by an automaton model.

Note that $\mathbf{S}$ is very close to strings considered as *term algebras*, that is, to $\langle \Sigma, \epsilon, (l_a)_{a\in\Sigma}\rangle$. It is of course well-known that the theory of arbitrary term algebras is decidable and admits QE [24]. However, adding the prefix relation is not necessarily a trivial addition: for arbitrary term algebras with prefix (subterm), only the *existential* theory is decidable, but the full theory is undecidable [30] (similar results hold for other orderings on terms [13]). The undecidability result of [30] requires at least one binary term constructor; our results indicate that in the simpler case of strings one recovers QE with the prefix relation.

We start with a result that gives a normal form for formulae of FO($\mathbf{S}$). Given a set S of strings , we let Tree(S) be the tree (i.e. the partially-ordered structure) generated by closing $S \cup \{\epsilon\}$ under $\sqcap$. In other words, Tree(S) is the poset $\langle \{x \sqcap y \mid x, y \in S \cup \{\epsilon\}\}, \prec\rangle$. (Note that for any set of strings $s_1, \ldots, s_k$, there are two indices $i, j \le k$ such that $s_1 \sqcap \ldots \sqcap s_k = s_i \sqcap s_j$.)

A *complete tree-order description* of a vector $\vec{w}$ of variables is the atomic diagram of Tree($\vec{w}$) in the language of $\epsilon, \preceq, \sqcap$. In other words, it is a specification of all the $\preceq$ relations that hold and do not hold in Tree($\vec{w}$).

For each $L \subseteq \Sigma^*$, let P_L be the set of pairs (x,y) of strings such that $x \preceq y$ and $y - x \in L$. The following lemma is obvious, since it is well-known that star-free sets are first-order definable on string models [25].

Lemma 1 *For each star free language* L*, there is a formula* $\varphi_L(x,y)$ *in* FO($\mathbf{S}$) *which defines* P_L. □

We now give a normal form result for FO($\mathbf{S}$).

Proposition 2 *Every formula* $\psi(\vec{x})$ *in* FO($\mathbf{S}$) *can be effectively transformed into an equivalent formula which is a disjunction of formulae of the form*

$$\gamma(\vec{x}) \wedge \delta(\vec{x})$$

where $\gamma(\vec{x})$ *is a complete tree-order description over* $\vec{x}$ *and* $\delta(\vec{x})$ *is a conjunction of formulae of the form* $\varphi_L(t(\vec{x}), t'(\vec{x}))$*, where* L *is star-free,* $t(\vec{x})$ *and* $t'(\vec{x})$ *are either* ϵ *or a term of the form* $x_i \sqcap x_j$*, and* $\gamma(\vec{x})$ *implies that* $t(\vec{x})$ *is an immediate successor of* $t'(\vec{x})$ *in the tree-order.*

Proof is by induction on the structure of ψ. □

Let $\mathbf{S}^+$ be the expansion of $\mathbf{S}$ to the signature that contains ϵ, $\sqcap$ and a binary predicate P_L for each star-free language L. Note that $\mathbf{S}^+$ is a *definable* expansion of $\mathbf{S}$, as all additional functions and predicates are definable. From the normal form we now immediately obtain:

Theorem 2 $\mathbf{S}^+$ *admits quantifier elimination.*

Remark. As mentioned above there is no need to nest the $\sqcap$-operator. Therefore, $\mathbf{S}^+$ can be turned into a relational signature that admits quantifier elimination as follows. For each star-free L let P'_L be the set of tuples (s_1, s_2, s_3, s_4) of strings for which $P_L(\sqcap(s_1,s_2), \sqcap(s_3,s_4))$. Note, that $\sqcap(s_1,s_2) \preceq \sqcap(s_3,s_4)$ can be expressed as $P_{\Sigma^*}(\sqcap(s_1,s_2), \sqcap(s_3,s_4))$. It is straightforward to check that this signature admits quantifier elimination. In the same way, the quantifier elimination results in the remainder of the paper can be turned into quantifier elimination results in a relational signature.

Note also that $\mathbf{S}^+$ could be considered as an expansion of $\mathbf{S}$ with either functions l_a or predicates L_a in the signature. In the latter case, predicates L_a are not needed as $L_a(x)$ iff $P_{\Sigma^* a}(\epsilon, x)$.

Another corollary of the normal form is that in the language of $\mathbf{S}$, it suffices to use only bounded quantification. That is, we introduce *bounded quantifiers* of

the form $\exists x \preceq y$ and $\forall x \preceq y$ (where $\exists x \preceq y\ \varphi$ means $\exists x\ x \preceq y \wedge \varphi$), and let $\mathrm{FO}_b(\mathbf{S})$ be the restriction of $\mathrm{FO}(\mathbf{S})$ to formulae $\varphi(y_1, \ldots, y_k)$ in which all quantifiers are of the form $Qx \preceq y_i$. From the normal form and the fact that each φ_L can be defined with bounded quantifiers, we obtain:

Corollary 1 $\mathrm{FO}_b(\mathbf{S}) = \mathrm{FO}(\mathbf{S})$. □

Finally, we characterize $\mathbf{S}$-definable subsets of Σ^* and $(\Sigma^*)^k$. Given a subset $R \subseteq (\Sigma^*)^k$ and a permutation π on $\{1, \ldots, k\}$, by $\pi(R)$ we mean the set $\{(s_{\pi(1)}, \ldots, s_{\pi(k)}) \mid (s_1, \ldots, s_k) \in R\}$.

Corollary 2

a) A language $L \subseteq \Sigma^*$ is definable in $\mathbf{S}$ iff it is star-free.

b) The class of relations definable over $\mathrm{FO}(\mathbf{S})$ is the minimal class containing the empty set, $\{\epsilon\}$, $\{a\}$ $a \in \Sigma$, $\preceq$, $\sqcap$, and closed under Boolean operations, Cartesian product, permutation, and the operation $*$ defined by $L_1 * L_2 = \{(s_1, s_1 \cdot s_2) \mid s_1 \in L_1, s_2 \in L_2\}$ for $L_1, L_2 \subseteq \Sigma^*$.

Proof. a) $\mathbf{S}^+$ formulae in one free variable are Boolean combinations of $P_L(\epsilon, x)$, for L star-free, and thus they define only star-free languages.

b) For one direction notice that ϵ, $\{a\}$, $\prec$, $\sqcap$ are definable in $\mathrm{FO}(\mathbf{S})$, and that $\mathrm{FO}(\mathbf{S})$ is closed under boolean operations, permutation and Cartesian product. The closure under $*$ is an easy consequence of Lemma 1 as $L_1 * L_2$ corresponds to $\{(x, y) \mid \varphi_{L_1}(\epsilon, x) \wedge \varphi_{L_2}(x, y)\}$. The other direction follows from the normal form. □

Note that the projection operation is not needed in the closure result above.

Automaton We now give an automaton model characterizing definability in $\mathrm{FO}(\mathbf{S})$. This automaton model corresponds exactly to the counter-free variant of *regular prefix automaton* as defined in [1].

Let us recall the definition of regular prefix automaton. Let A be a finite non-deterministic automaton on strings with state set Q, transition relation δ and initial state q_0. We construct from A an automaton $\hat{A} = (\Sigma, Q, q_0, F, \delta)$ accepting n-tuples $\vec{w} = (w_1, \cdots, w_n)$ of strings in the following way. F is a subset of Q^n which denotes the accepting states of $\hat{A}$. Let $\mathit{prefix}(\vec{w})$ be the set of all prefixes of all w_i. A *run* of $\hat{A}$ over $\vec{w}$ is a mapping h from $\mathit{prefix}(\vec{w})$ to Q which assigns to every node $\alpha \in \mathit{prefix}(\vec{w})$ a state $q \in Q$ such that $h(\epsilon) = q_0$ and, $\beta = l_a(\alpha)$ implies $h(\beta) \in \delta(h(\alpha), a)$. The run is accepting if $(h(w_1), \cdots, h(w_n)) \in F$. The n-tuple $\vec{w}$ is accepted by $\hat{A}$ if there is an accepting run of $\hat{A}$ over $\vec{w}$. See [1] for more details.

For each finite non-deterministic automaton A the corresponding automaton $\hat{A}$ is called *regular prefix automaton* (RPA). The subset of $(\Sigma^*)^n$, $n \in \mathbb{N}$, it defines is called a *regular prefix relation* (RPR).

If the automaton A is counter-free then we say that the corresponding automaton $\hat{A}$ is counter-free (CF-PA). The following shows that the relations definable in $\mathrm{FO}(\mathbf{S})$ are exactly those recognizable by a CF-PA.

Proposition 3 *A relation is definable in $\mathrm{FO}(\mathbf{S})$ if and only if it is definable by a counter-free prefix automaton.* □

It should be noted that $\mathrm{FO}(\mathbf{S})$ can also be characterized by means of counter-free deterministic bottom-up automata.

VC-dimension and Isolation In addition to quantifier elimination, $\mathbf{S}$ has some further model-theoretic properties that distinguish it from $\mathbf{S}_{\mathrm{len}}$.

Proposition 4 $\mathrm{Th}(\mathbf{S})$ *has the strong isolation property.* □

As a corollary of the isolation property, we prove that, unlike for $\mathbf{S}_{\mathrm{len}}$, the definable families for $\mathbf{S}$ are learnable. First, we need the following.

Proposition 5 *Let M be a model with the isolation property. Then its definable families have finite VC-dimension.*

We give two proofs of this result in the full version: one is a complexity-theoretic argument, the other model-theoretic. □

It follows that the model $\mathbf{S}$, unlike $\mathbf{S}_{\mathrm{len}}$, has learnable definable families.

Corollary 3 *Every definable family in $\mathbf{S}$ has finite VC-dimension.* □

5 Star-free algebra based on $\mathbf{S}_{\mathrm{left}}$

We now study an example of a star-free algebra, one where the n-ary relations in the algebra are more complex than those definable over $\mathbf{S}$. Recall that

$\mathbf{S}_{\text{left}} = \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma}, (f_a)_{a\in\Sigma}\rangle$; that is, in this structure one can add characters on the left as well as on the right.

Without the prefix relation, this structure was studied in [27], where a quantifier-elimination result was proved, by extending quantifier-elimination for term algebras (in fact [27] showed that term algebras with queues admit QE). However, as in the case of $\mathbf{S}$, which differs from strings as terms algebras in that it has the prefix relation, here, too, the prefix relation complicates things considerably.

We start with an easy observation that $\text{FO}(\mathbf{S}_{\text{left}})$ expresses more relations that $\text{FO}(\mathbf{S})$. Indeed, the graph of f_a, $F_a = \{(x, a\cdot x) \mid x \in \Sigma^*\}$ is not expressible in $\text{FO}(\mathbf{S})$, which can be shown by a simple game argument. More precisely, given a number k of rounds, let $n = 2^k+1$ and consider the game on the tuples $(0^n, 10^n)$ and $(0^{n+1}, 10^n)$. By Corollary 1 it is sufficient to play on the prefixes of the participating strings. The duplicator has a trivial winning strategy on the strings 10^n and a well-known winning strategy on 0^n versus 0^{n+1}.

Let $\mathbf{S}^+_{\text{left}}$ be the extension of $\mathbf{S}_{\text{left}}$ with the same (definable) functions and predicates we added to $\mathbf{S}^+$ (that is, a constant ϵ for the empty string, the binary function $\sqcap$ for the longest common prefix, the predicate $P_L(x,y)$ for each star-free language L), and the unary function $x \mapsto x - a$, for each $a \in \Sigma$ (which is also definable).

Theorem 3 $\mathbf{S}^+_{\text{left}}$ *admits quantifier elimination.*

Proof sketch. Let $\Omega_{\mathbf{S}^+}$ and $\Omega_{\mathbf{S}^+_{\text{left}}}$ be the first-order signatures of $\mathbf{S}^+$ and $\mathbf{S}^+_{\text{left}}$. Let M be an ω-saturated model over $\Omega_{\mathbf{S}^+_{\text{left}}}$ elementary equivalent to $\mathbf{S}^+_{\text{left}}$. It suffices to prove quantifier elimination in M. Note that M can have both finite and infinite strings. To prove QE, we must show that every two tuples of elements of M that have the same *atomic* type, have the same type. Define a *nice term* of $\Omega_{\mathbf{S}^+_{\text{left}}}$ as a term of the form $t(x) = x - a + b$, where a and b are finite strings. Given two tuples $\vec{c}$ and $\vec{d}$ of the same length over M, define two relations on them:

- $\vec{c} \equiv \vec{d}$ iff for all sequences $i_1, \ldots, i_k$ from $\{1, \ldots, n\}$ (where n is the length of $\vec{c}$) and all sequences $t_1, \ldots, t_k$ of nice terms:
$$\begin{aligned} & atp_{\mathbf{S}^+}(t_1(c_{i_1}), \ldots, t_k(c_{i_k})) \\ = \; & atp_{\mathbf{S}^+}(t_1(d_{i_1}), \ldots, t_k(d_{i_k})) \end{aligned}$$
- $(c', \vec{c}) \equiv_1 (d', \vec{d})$ iff for all sequences $i_1, \ldots, i_k$ from $\{1, \ldots, n\}$ and all sequences $t_1, \ldots, t_k$ of nice terms:
$$\begin{aligned} & atp_{\mathbf{S}^+}(c', t_1(c_{i_1}), \ldots, t_k(c_{i_k})) \\ = \; & atp_{\mathbf{S}^+}(d', t_1(d_{i_1}), \ldots, t_k(d_{i_k})) \end{aligned}$$

Of course, $(c', \vec{c}) \equiv (d', \vec{d})$ implies $(c', \vec{c}) \equiv_1 (d', \vec{d})$, as the identity is a nice term. We then prove the main lemma, which shows that these two relations coincide; that is, if $(c', \vec{c}) \equiv_1 (d', \vec{d})$, then also $(c', \vec{c}) \equiv (d', \vec{d})$.

Using this, we show that $\equiv$ has the back-and-forth property in M (which is actually stronger than what is needed for quantifier-elimination). The theorem follows from the lemma, as each type of the form $atp_{\mathbf{S}^+}(t_1(c_{i_1}), \ldots, t_k(c_{i_k}))$ is also an atomic type of $\mathbf{S}^+_{\text{left}}$. Hence, the atomic types determine the types. For details, see the full version [6]. □

From the previous theorem we get the following corollaries. First, the back-and-forth property of $\equiv_1$ gives us the following normal form for $\text{FO}(\mathbf{S}^+_{\text{left}})$ formulae.

Corollary 4 *For every* $\text{FO}(\mathbf{S}_{\text{left}})$ *formula* $\rho(x, \vec{y})$ *there is an* $\text{FO}(\mathbf{S})$ *formula* $\rho'(x, \vec{z})$ *and a finite set of nice* $\mathbf{S}^+_{\text{left}}$ *terms* $\vec{t}$ *such that*

$$\forall x \vec{y}\ \rho(x, \vec{y}) \leftrightarrow \rho'(x, \vec{t}(\vec{y}))$$

holds in $\mathbf{S}_{\text{left}}$. □

Then Corollary 4 for the empty tuple $\vec{y}$ and Corollary 2 imply:

Corollary 5 *Subsets of* Σ^* *definable over* $\mathbf{S}_{\text{left}}$ *are precisely the star-free languages.* □

For formulae in the language of $\mathbf{S}_{\text{left}}$ (as opposed to $\mathbf{S}^+_{\text{left}}$), we can show that bounded quantification suffices, although the notion of bounded quantification is slightly different here from that used in the previous section. Let $N_p(s)$ be the prefix-closure of $\{s - s_1 + s_2 \mid |s_1|, |s_2| \le p\}$. Clearly $N_p(s)$ is definable from s over $\mathbf{S}_{\text{left}}$. We then define $\text{FO}_*(\mathbf{S}_{\text{left}})$ as the class of $\text{FO}(\mathbf{S}_{\text{left}})$ formulae $\varphi(\vec{x})$ in which all quantification is of the form $\exists z \in N_p(x_i)$ and $\forall z \in N_p(x_i)$, where x_i is a free variable of φ and $p \ge 0$ arbitrary.

Corollary 6 $\text{FO}_*(\mathbf{S}_{\text{left}}) = \text{FO}(\mathbf{S}_{\text{left}})$. □

Isolation and VC-dimension We now show that the results about isolation and VC-dimension extend from $\mathbf{S}$ to $\mathbf{S}_{\text{left}}$.

Proposition 6 $\text{Th}(\mathbf{S}_{\text{left}})$ *has the isolation property.* □

Since the argument for corollary 3 actually shows that isolation implies finite VC-dimension, we conclude:

Corollary 7 *Every definable family in* $\mathbf{S}_{\text{left}}$ *has finite VC-dimension.* □

6 Regular algebra extending S

The previous sections presented star-free algebras with attractive properties. We now give an example of a regular algebra that has significantly *less* expressive power than the rich structure $\mathbf{S}_{\text{len}}$, and which shares some of the nicer properties of the star-free algebras in the previous sections.

This algebra can be obtained by considering two possible ways of extending FO(**S**): the first is by adding the predicates P_L for all *regular* languages L; that is, predicates $P_L(x,y)$ which hold for $x \preceq y$ such that $y - x \in L$, where L is a regular language. The second extension is by using monadic-second order logic instead of only first-order logic. It turns out that these extensions define exactly the same algebra. We show this, and also show that the resulting regular algebra shares the QE and VC-dimension properties of the star-free algebras defined previously.

Let $\mathbf{S}_{\text{reg}} = \langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma}, (P_L)_{L \text{ regular}} \rangle$. Since it defines arbitrary regular languages in Σ^*, it is a proper extension of **S**. Every FO($\mathbf{S}_{\text{reg}}$)-definable set is definable over $\mathbf{S}_{\text{len}}$, because the predicates P_L are definable in $\mathbf{S}_{\text{len}}$ (the easiest way to see this is by using the characterization of $\mathbf{S}_{\text{len}}$ definable properties via letter-to-letter automata). Thus, we have:

Proposition 7 *Subsets of Σ^* definable over $\mathbf{S}_{\text{reg}}$ are precisely the regular languages.* □

Let $\mathbf{S}^+_{\text{reg}}$ be the extension of $\mathbf{S}_{\text{reg}}$ with ϵ and $\sqcap$. Most of the results about **S** and $\mathbf{S}^+$ from Section 4 can be straightforwardly lifted to $\mathbf{S}_{\text{reg}}$ and $\mathbf{S}^+_{\text{reg}}$. For example, the normal form Proposition 2 holds for $\mathbf{S}_{\text{reg}}$ if one replaces "star-free" with "regular": the proof given in Section 4 applies verbatim. From this normal form we immediately obtain:

Theorem 4 $\mathbf{S}^+_{\text{reg}}$ *admits quantifier elimination.* □

The normal form result also shows that neither the functions f_a nor the predicate el are definable in $\mathbf{S}_{\text{reg}}$ (the former can also be seen from the fact that $\mathbf{S}_{\text{reg}}$ has QE in a signature of bounded arity, and $\mathbf{S}_{\text{len}}$ does not; for inexpressibility of f_a it suffices to apply the normal form results to pairs of strings of the form $(1 \cdot 0^k, 0^k)$). One can also show, as in the case of **S**, that bounded quantification over prefixes is sufficient.

Our next aim is to show that FO($\mathbf{S}_{\text{reg}}$) gives us exactly the same algebra of definable sets as MSO(**S**).

Notice first that each relation definable in FO($\mathbf{S}_{\text{reg}}$) is already definable in MSO(**S**) because each predicate P_L is definable in MSO. We will show in the following that the converse implication also holds.

The proof relies on a lemma which essentially shows that the monadic second-order type of a tuple of strings only depends on its tree-order type and the monadic second-order types of the paths between the strings and their common prefixes.

For a sequence $\vec{a} = (a_1, \ldots, a_n)$ of strings, let $T_{\vec{a}}$ be the structure $\langle \Sigma^*, \preceq, (L_a)_{a\in\Sigma}, \vec{a} \rangle$.

For each string $w \in \Sigma^*$, let $\mathcal{I}_w$ be the finite structure $\langle I_w, <, (R_a)_{a\in\Sigma}, 1, |w| \rangle$ where I_w is $\{1, \ldots, |w|\}$, $<$ is the usual order and, for each $a \in \Sigma$, R_a is the set of all positions of w that carry the letter a. For two strings $u, v \in \Sigma^*$, we write $u \equiv^s_k v$ if $\mathcal{I}_u \equiv_{\text{MSO}_k} \mathcal{I}_v$.

Lemma 2 *For each $k > 0$, there is $k' > 0$ such that the following holds. Let $\vec{a} = (a_1, \ldots, a_n), \vec{b} = (b_1, \ldots, b_n)$ be sequences of strings for which there is a tree isomorphism h : Tree($\vec{a}$) $\to$ Tree($\vec{b}$) such that*

(i) for each $i \in \{1, \ldots, n\}$, $h(a_i) = b_i$, and

(ii) whenever u is the immediate predecessor of v in Tree($\vec{a}$) then $v - u \equiv^s_k h(v) - h(u)$.

Then $T_{\vec{a}} \equiv_{\text{MSO}_k} T_{\vec{b}}$. □

As both conditions (i) and (ii) of the Lemma are expressible in FO($\mathbf{S}_{\text{reg}}$), we obtain:

Theorem 5 FO($\mathbf{S}_{\text{reg}}$) = MSO(**S**). □

The bounded monadic second-order quantifier $\exists X \preceq y$ is defined as follows. A formula $\exists X \preceq y\ \varphi$ holds if and only if $\exists X(\forall x X(x) \to x \preceq y) \wedge \varphi$ holds. We define $\text{MSO}_b(\mathbf{S})$ by binding all first-order and monadic second-order quantifiers.

From Theorem 5 we can easily derive the following corollaries.

Corollary 8

- $\text{MSO}_b(\mathbf{S}) = \text{MSO}(\mathbf{S})$
- *Subsets of Σ^* definable in* MSO(**S**) *are exactly the regular languages.*

Automata model, isolation, and VC dimension It was proved in [1] that Regular Prefix Relations (RPR) (those definable by Regular Prefix Automata (RPA), introduced in Section 4) are exactly those definable in MSO(**S**). Thus Theorem 5 together with the results of [1] gives a new characterization of FO($\mathbf{S}_{\text{reg}}$).

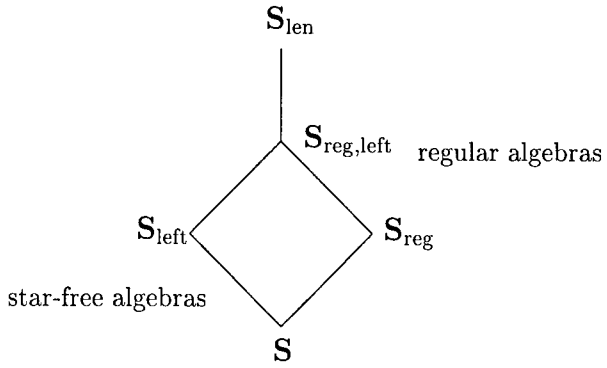


Figure 1. *Relationships between* $\mathbf{S}, \mathbf{S}_{\text{left}}, \mathbf{S}_{\text{reg}}, \mathbf{S}_{\text{reg,left}}$, *and* $\mathbf{S}_{\text{len}}$.

Corollary 9 *The relations definable in* $\text{FO}(\mathbf{S}_{\text{reg}})$ *are exactly the RPR relations. Thus each relation definable in* $\text{FO}(\mathbf{S}_{\text{reg}})$ *is recognizable by a RPA.* □

The proof of the isolation property for $\mathbf{S}$ (Proposition 4) is unaffected by the change from star-free P_L to regular P_L. Thus, we obtain:

Corollary 10 $\text{Th}(\mathbf{S}_{\text{reg}})$ *has the isolation property, and definable families of* $\mathbf{S}_{\text{reg}}$ *have finite VC-dimension.* □

7 Regular algebra extending $\mathbf{S}_{\text{left}}$

We now give a final example of a regular algebra. Let $\mathbf{S}_{\text{reg,left}}$ be the common expansion of $\mathbf{S}_{\text{left}}$ and $\mathbf{S}_{\text{reg}}$, that is, $\langle \Sigma^*, \preceq, (l_a)_{a\in\Sigma}, (f_a)_{a\in\Sigma}, (P_L)_{L \text{ regular}} \rangle$. Since $\mathbf{S}_{\text{reg}}$ cannot express the functions f_a, and $\mathbf{S}_{\text{left}}$ cannot define arbitrary regular sets, we see that $\mathbf{S}_{\text{reg,left}}$ is a proper expansion of $\mathbf{S}_{\text{reg}}$ and $\mathbf{S}_{\text{left}}$. Furthermore, all $\mathbf{S}_{\text{reg,left}}$-definable sets are $\mathbf{S}_{\text{len}}$-definable; the finiteness of VC dimension for $\mathbf{S}_{\text{reg,left}}$, shown below, implies that this containment is proper, too.

Let $\mathbf{S}^+_{\text{reg,left}}$ be the common expansion of $\mathbf{S}^+_{\text{left}}$ and $\mathbf{S}_{\text{reg}}$, that is, the expansion of $\mathbf{S}_{\text{reg,left}}$ with ϵ and $\sqcap$. The techniques of the previous sections can be used to show the following:

Theorem 6 $\mathbf{S}^+_{\text{reg,left}}$ *has quantifier-elimination. Furthermore,* $\text{Th}(\mathbf{S}_{\text{reg,left}})$ *has the isolation property, and definable families in* $\mathbf{S}_{\text{reg,left}}$ *have finite VC-dimension.* □

Similarly to $\mathbf{S}_{\text{left}}$, we derive from the proof of Theorem 6 the following normal form for $\mathbf{S}_{\text{reg,left}}$ formulae:

Corollary 11 *For every* $\text{FO}(\mathbf{S}_{\text{reg,left}})$ *formula* $\rho(x, \vec{y})$ *there is an* $\text{FO}(\mathbf{S}_{\text{reg}})$ *formula* $\rho'(x, \vec{z})$ *and a finite set of nice* $\mathbf{S}^+_{\text{left}}$ *terms* $\vec{t}$ *such that*

$$\forall x \vec{y}\ \rho(x, \vec{y}) \leftrightarrow \rho'(x, \vec{t}(\vec{y}))$$

holds in $\mathbf{S}_{\text{reg,left}}$. □

We conclude this section with a remark showing that arithmetic properties definable in structures $\mathbf{S}, \mathbf{S}_{\text{left}}, \mathbf{S}_{\text{reg}}, \mathbf{S}_{\text{reg,left}}$ are weaker than those definable in $\mathbf{S}_{\text{len}}$. As we mentioned earlier, under the binary encoding, $\mathbf{S}_{\text{len}}$ gives us an extension of Presburger arithmetic; namely, it defines $+$ and V_2, where $V_2(x)$ is the largest power of 2 that divides x. But even $\mathbf{S}_{\text{reg,left}}$ is much weaker:

Proposition 8 *Neither successor, nor order, nor addition, are definable in* $\mathbf{S}_{\text{reg,left}}$ *(and hence in* $\mathbf{S}, \mathbf{S}_{\text{reg}}, \mathbf{S}_{\text{left}}$*).* □

8 Conclusion

There has been significant interest in theoretical computer science in understanding the structure of the regular languages, and in identifying subclasses of the regular languages that have special properties [29, 28]. Our work can be seen as an extension of this program, where we consider subclasses of the regular n-ary relations rather than the regular sets. In our approach, however, we do not focus on properties that hold of one particular regular relation by itself, but rather look at some desirable properties of a whole algebra of relations lying within the structure $\mathbf{S}_{\text{len}}$.

We have shown a sharp contrast between the behavior of the full algebra of regular relations of $\mathbf{S}_{\text{len}}$, and those of various submodels such as $\mathbf{S}$, $\mathbf{S}_{\text{left}}$, $\mathbf{S}_{\text{reg}}$, and $\mathbf{S}_{\text{reg,left}}$. We show that the latter are more tractable in many respects. Furthermore, we show that the behavior of an algebra of relations is not at all determined by

the one-dimensional sets (subsets of Σ^*) in the algebra: for example, one can have fairly complex binary relations definable, yet still maintain the property that all definable subsets of Σ^* are star-free. Figure 1 summarizes the relationships between the star-free and regular algebras we considered here.

A key question is how many relations one can add to the models $\mathbf{S}_{\text{left}}$ or $\mathbf{S}_{\text{reg}}$ and still have the attractive properties like QE and finite VC-dimension. Is there a model that is somehow maximal with respect to these properties? We would very much like to know the answer to this question. There are also several natural candidate models that would seem amenable to the approach taken here, and where one would expect the same results to go through: for example, if one allows the operation concatenating a fixed sequence "in the middle" of a string, rather than on the left or on the right, is the resulting model still tractable?

References

[1] D. Angluin, D. N. Hoover. Regular prefix relations. *Mathematical Systems Theory* 17(3),167–191,1984.

[2] M. Anthony and N. Biggs. *Computational Learning Theory.* Cambridge Univ. Press, 1992.

[3] O. Belegradek, A. Stolboushkin, M. Taitslin. Extended order-generic queries. *Annals of Pure and Applied Logic* 97 (1999), 85–125.

[4] A. Blumer, A. Ehrenfeucht, D. Haussler, M. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM* 36 (1989), 929–965.

[5] M. Benedikt, L. Libkin. Relational queries over interpreted structures. *J. ACM* 47 (2000), 644–680.

[6] M. Benedikt, L. Libkin, T. Schwentick, L. Segoufin. A model-theoretic approach to regular string relations. INRIA Technical Report, 2000. Available at http://www-rocq.inria.fr/verso/publications/.

[7] M. Benedikt, L. Libkin, T. Schwentick, L. Segoufin. String operations in query languages. In *PODS'01*, pages 183–194.

[8] J.R. Büchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik Grundl. Math.* 6 (1960), 66–92.

[9] A. Blumensath and E. Grädel. Automatic structures. In *LICS'00*, pages 51–62.

[10] V. Bruyère, G. Hansel, C. Michaux, R. Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc.* 1 (1994), 191–238.

[11] C.C. Chang and H.J. Keisler Model Theory. North Holland, 1990.

[12] G. Cherlin and F. Point. On extensions of Presburger arithmetic. In *Proc. 4th Easter Model Theory Conf.*, Humboldt Univ. Berlin, 1986.

[13] H. Comon, R. Treinen. The first-order theory of lexicographic path orderings is undecidable. *TCS* 176 (1997), 67–87.

[14] E. Dantsin, A. Voronkov. Expressive power and data complexity of query languages for trees and lists. In *PODS'2000*, pages 157–165.

[15] C. Elgot and J. Mezei. On relations defined by generalized finite automata. *IBM J. Res. Develop.* 9 (1965), 47–68.

[16] D. Epstein et al. *Word Processing in Groups.* Jones and Bartlett Publ., 1992.

[17] J. Flum and M. Ziegler. Pseudo-finite homogeneity and saturation. Preprint, Freiburg University, 1998.

[18] C. Frougny and J. Sakarovitch. Synchronized rational relations of finite and infinite words. *TCS* 108 (1993), 45–82.

[19] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation* 140 (1998), 26–81.

[20] G. Grahne, M. Nykänen, E. Ukkonen. Reasoning about strings in databases. *JCSS* 59 (1999), 116–162.

[21] D. Harel. Towards a theory of recursive structures. In *MFCS'98*, pages 36–53.

[22] B. Khoussainov and A. Nerode. Automatic presentations of structures. In *LCC'94*, pages 367–392.

[23] G. Kuper, L. Libkin, J. Paredaens, editors. *Constraint Databases.* Springer, 2000.

[24] A. Maĺcev. On the elementary theories of locally free universal algebras. *Soviet Math. Doklady* 2 (1961), 768–771.

[25] R. McNaughton and S. Papert. *Counter-Free Automata.* MIT Press, 1971.

[26] C. Michaux, R. Villemaire. Open questions around Büchi and Presburger arithmetics. In *Logic: From Foundations to Applications*, Oxford Univ. Press, 1996, pages 353–383.

[27] T. Rybina, A. Voronkov. A decision procedure for term algebras with queues. *LICS'2000*, pages 279–290.

[28] H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhäuser, 1994.

[29] W. Thomas. Languages, automata, and logic. *Handbook of Formal Languages, Vol. 3*, Springer, 1997.

[30] K. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *J. ACM* 34 (1987), 492–510.

Author Index

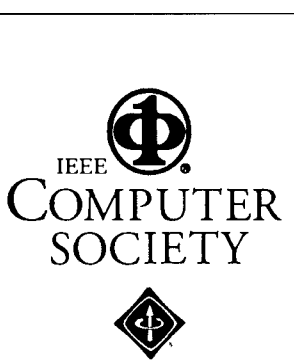
Abiteboul, S. 379
Adler, M. 197
Alon, N. 421
Altenkirch, T. 303
Alur, R. 291
Appel, A. 247
Arnold, A. 157
Asarin, E. 269
Avigad, J. 139
Barrett, C. 29
Barrington, D. 187
Benedikt, M. 431
Berger, M. 311
Blanqui, F. 9
Bouajjani, A. 269, 399
Bruns, G. 409
Cook, S. 177
Das, S. 51
de Alfaro, L. 279
Dill, D. 29, 51
Dybjer, P. 303
Escardó, M. 115
Fiore, M. 93
Ganzinger, H. 81
Godefroid, P. 409
Godoy, G. 38
Gurevich, Y. 129
Henzinger, T. 279
Hofmann, M. 303
Honda, K. 311
Huuskonen, T. 167
Hyttinen, T. 167
Immerman, N. 187, 197
Janin, D. 347
Jeffrey, A. 323
Kirousis, L. 71
Kolaitis, P. 71
Kolokolova, A. 177
Kozen, D. 259
Kupferman, O. 389
La Torre, S. 291
Laird, J. 105
Lange, M. 357
Lautemann, C 187
Lenzi, G. 157, 347
Levitt, J. 29
Libkin, L. 431
Majumdar, R. 279
Manolios, P. 366
Marcinkowski, J. 157
Milo, T. 421
Mitchell, J. 3
Muscholl, A. 399
Neven, F. 421
Nieuwenhuis, R. 38
Pfenning, F. 221
Ramanathan, A. 3
Rueβ, H. 19
Salibra, A. 334
Scedrov, A. 3
Schweikardt, N. 187
Schwentick, T. 431
Scott, P. 303
Segoufin, L. 431
Shankar, N. 19
Simpson, A. 115
Stirling, C. 357
Stoller, S. 61
Stump, A. 29
Suciu, D. 421
Szwast, W. 147
Teague, V. 3
Tendera, L. 147
Terui, K. 209
Thérien, D. 187
Tiuryn, J. 259
Touili, T. 399
Trefler, R. 366
Turi, D. 93
Vardi, M. 389
Vianu, V. 421
Xi, H. 231
Yoshida, N. 311

Notes

Notes

Notes

Notes



Press Operating Committee

Chair
Mark J. Christensen
Independent Consultant

Editor-in-Chief
Mike Williams
Department of Computer Science, University of Calgary

Board Members

Roger U. Fujii, *Vice President, Logicon Technology Solutions*
Richard Thayer, *Professor Emeritus, California State University, Sacramento*
Sallie Sheppard, *Professor Emeritus, Texas A&M University*
Deborah Plummer, *Group Managing Editor, Press*

IEEE Computer Society Executive Staff

Anne Marie Kelly, *Acting Executive Director*
Angela Burgess, *Publisher*

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at *http://computer.org* for a list of products.

IEEE Computer Society Proceedings

The IEEE Computer Society also produces and actively promotes the proceedings of more than 160 acclaimed international conferences each year in multimedia formats that include hard and softcover books, CD-ROMs, videos, and on-line publications.

For information on the IEEE Computer Society proceedings, please e-mail to csbooks@computer.org or write to Proceedings, IEEE Computer Society, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1-714-821-8380. Fax +1-714-761-1784.

Additional information regarding the Computer Society, conferences and proceedings, CD-ROMs, videos, and books can also be accessed from our web site at *http://computer.org/cspress*

Revised April 20, 2001