

DTIC FILE COPY

RADC-TR-89-108
Final Technical Report
August 1989



4

AD-A213 504

MASC: MULTIPROCESSOR ARCHITECTURE FOR SYMBOLIC PROCESSING

UNISYS Paoli Research Center

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order No. 5475

DTIC
ELECTE
OCT 06 1989
S D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 10 6 049

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-108 has been reviewed and is approved for publication.

APPROVED: Robert L. Kaminski

ROBERT L. KAMINSKI
Project Engineer

APPROVED: Raymond P. Urtz, Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

MASC: MULTIPROCESSOR ARCHITECTURE FOR SYMBOLIC PROCESSING

William C. Hopkins
Tom M. Blenko
Keith Cassell
Judy Pohl Clark
Joel B. Coltoff

Daniel R. Corpron
Massoud Farhang
Lynette Hirschman
Donald P. McKay
Robert C. Smith

Contractor: UNISYS Paoli Research Center
Contract Number: F30602-86-C-0093
Effective Date of Contract: 27 Mar 86
Contract Expiration Date: 30 Sep 88
Short Title of Work: MASC: Multiprocessor Architecture
for Symbolic Processing
Program Code Number: 53910A
Period of Work Covered: Mar 86 - Sep 88

Principal Investigator: William C. Hopkins
Phone: (215) 648-7578

Project Engineer: Robert L. Kaminski
Phone: (315) 330-2925

Approved for Public Release; Distribution Unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Robert L. Kaminski RADC (COTC) Griffiss AFB NY 13441-5700, under Contract F30602-86-C-0093.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Date	
Project Number	PS
Contract Number	CF
Report Number	AI

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			Approved for public release; distribution unlimited.		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-106		
6a. NAME OF PERFORMING ORGANIZATION UNISYS Paoli Research Center		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)		
6c. ADDRESS (City, State, and ZIP Code) P.O. Box 517 Paoli PA 19301			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency		8b. OFFICE SYMBOL (If applicable) ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-86-C-0093		
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd Arlington VA 22209			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 62301E	PROJECT NO. E475	TASK NO. 01
			WORK UNIT ACCESSION NO. 01		
11. TITLE (Include Security Classification) MASC: MULTIPROCESSOR ARCHITECTURE FOR SYMBOLIC PROCESSING					
12. PERSONAL AUTHOR(S) William C. Hopkins, Tom M. Blenko, Keith Cassell, Judy Pohl Clark, Joel B. Coltoff, Daniel R. Corpron, Massoud Farhang, Lynette Hirschman, Donald P. McKay, Robert C. Smith					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM Mar 86 TO Sep 88		14. DATE OF REPORT (Year, Month, Day) August 1989	
				15. PAGE COUNT 384	
16. SUPPLEMENTARY NOTATION N/A					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Artificial Intelligence		
12	05		Functional Programming		
			Parallel Processing		
			Compilation of Logic		
			Simulation		
			Application Parallelism		
			Partial Evaluation		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Reports result in simulation of parallelism in Artificial Intelligence applications, specifically natural language parsing, in the compilation of logic programs to functional form, and in the integration of logic and functional programming.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIEDUNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Robert L. Kaminski			22b. TELEPHONE (Include Area Code) (315) 330-2925		22c. OFFICE SYMBOL RADC (COTC)

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Summary

The MASC program addresses the support of symbolic computation for Artificial Intelligence (AI), as part of the DARPA Strategic Computing Program. The work has been aimed at the fundamental questions of choosing an appropriate programming model for AI and finding efficient implementation techniques to support it on multiprocessor systems.

The Strategic Computing Program is motivated by the need to provide real-time response by AI subsystems in planned or contemplated DoD systems. Our efforts have directly supported this goal, contributing fundamental results in language design, language implementation, and application parallelism. The work has been driven by the needs of real AI applications, drawn from our substantial experience in natural language processing, knowledge representation, and expert systems. The technologies we have developed are broadly applicable, and several specific recommendations for their further development and application are presented.

The most profound result is the design and demonstration of a powerful technique for the compilation of logic programs to applicative form. This brings the power of compilation and optimization techniques for functional programming to bear on logic programming languages. We recommend that these technology be extended and applied to current research efforts in language design for very high level programming and program prototyping. Other results and recommendations are contained in the body of the report.

Keywords: natural processing, (K-P)

Preface

The work described in this report is the result of a collaboration among a diverse group of researchers. It draws heavily on the existing state of the art in AI technology, in functional language implementation techniques, and in the theory of logic programming; these debts are acknowledged in the references to published work. The functional language research group at Yale University, led by Dr. Paul Hudak, has, as part of the program effort, had a special role in the research; their contributions and insights have proven invaluable throughout the program.

The structure of this report is heirarchical. Each of the eight subsections and four appendices has its own table of contents and, if appropriate, lists of figures, tables, and references. The reader may refer to the initial pages of each section for a detailed description of its structure and contents.

Finally, I wish to acknowledge the technical contributions and support of the Unisys Paoli Research Center staff who made this program work. Their tireless efforts and technical imagination have made it possible to report success in several important areas. Theirs is the credit for the good work that we report. In addition to the listed authors of this report, the list includes Clarke Arnold, Rich Fritzson, Maggie Heine-man, and Howard Rosenshine. Errors and deficiencies, of course, remain the responsibility of the Principal Investigator.

William C. Hopkins
Principal Investigator

Contents

Section	Title	Pages
1	Overview of the MASC Program	9
2	Definition of Juniper-B1	49
3	Compiling Logic Programs to Applicative Form	41
4	GWAM: A Gadgetized Variant of the WAM	33
5	Or-Parallel Speed-Up in Natural Language Processing	19
6	Concurrency Simulation by Abstract Interpretation	17
7	The Virtual Computation Recorder	30
8	Directions for Future Research	6

Appendices

A	Issues in the Definition of Juniper	55
B	Small Logic Programs to Test Juniper	49
C	Juniper-B1 Unification	22
D	Partial Evaluation of Juniper-B1 Programs	8

Section 1

Overview of the MASC Program

by

**William C. Hopkins
Principal Investigator**

Table of Contents

Section	Title	Page
1.1	Orientation and Assumptions	1-1
1.2	Summary of Major Results	1-2
1.3	Juniper, a Logic + Functional Programming Language	1-2
1.4	Compiling Logic Programs to Applicative Form	1-3
1.5	Gadget Technology applied to the Warren Abstract Machine	1-4
1.6	Partial Evaluation of Declarative Programs	1-4
1.7	Parallelism in the PUNDIT Parser	1-5
1.8	Juniper-B1 Application Programs	1-6
1.9	Simulation of Concurrency by Abstract Interpretation	1-6
1.10	Future Directions	1-7

Section 1

Overview of the MASC Program

The Unisys Multi-processor Architecture for Symbolic Computation (MASC) program proposed to address the problem of architectural support of parallel symbolic computation, particularly artificial intelligence applications, both fundamentally and comprehensively. The program originally spanned the range from applications to hardware implementations of multiprocessor symbolic architectures. During the program's life, the focus narrowed to the upper levels of the spectrum: AI applications, languages and programming models for the applications, and evaluation and compilation models for the languages. We are pleased to report that we have provided fundamental and significant results in these areas.

1.1. Orientation and Assumptions

The program has maintained several important premises, which must be clearly understood to establish the context within which we have worked. The first of these is a strong commitment to addressing the application area as the Paoli Research Center understands it. The program has drawn upon the PRC staff for input and assistance in applications and programming models, building on their expertise in AI application design and symbolic program development. This practical orientation ensures that this research can be coupled to practice and to the support of large-scale applications.

The second premise is that logic programming (LP) is an appropriate and desirable model for AI applications. LP languages provide a number of features that support AI models, including logic variables and unification (supporting powerful pattern-matching and incremental specification of data structures), non-determinacy (allowing multiple solutions) and the reversibility of relations (supporting inference independent of pre-determined input and output parameters). Our original focus was on combining logic and functional programming paradigms, in order to exploit both the fine-grained parallelism and optimisation techniques in functional languages, as well as the coarser-grained and- and or-parallelism implicit in LP languages. This work led us to a powerful technique for embedding logic variables in a functional language, based on higher-order unification.

The third premise is the commitment to minimising the programmer's involvement in the question of parallelism. We believe that programming is sufficiently difficult without introducing another dimension to the process. Annotations to identify concurrency are the limit to what we are willing to ask of the programmer. This implies powerful program analysis techniques, which are greatly simplified in languages that avoid "anti-parallel" or sequentialising features.

Finally, we have attempted to maintain a global picture of the systems that may result from our research, optimising the overall performance of the system rather than

concentrating on one element to the detriment of others. Thus, we have found the need to balance program analysis and optimization techniques with the blind exploitation of parallelism.

1.2. Summary of Major Results

- An application suite, consisting of both logic and functional programs. For benchmarking purposes, there are a total of 15 programs, some in both logic and functional version, testing specific language features. In addition, there are 2 AI applications, drawn from the ongoing work in AI and natural language processing at PRC.
- Juniper-B1, a combined logic/functional programming language, combining the applicative, lazy evaluation semantics of a functional language with much of the functionality of a logic programming language, including evaluable functions combined with logic variables.
- Compilation of logic to (almost) applicative functional form via *gadgets*. By representing logic variables as higher-order functions, logic can be compiled into a slightly enhanced functional language, which supports powerful optimisations. Our results showed that there was no significant performance degradation resulting from the use of gadget technology in the context of a conventional Prolog implementation, while offering significant new opportunities for logic program optimisation.
- Program optimisation. We developed a partial evaluation system for the functional subset of Juniper-B1 that is invoked within the compiler, providing substantial optimisation of function applications.
- Utility of or-parallelism for a large-scale application. We obtained simulation results indicated a 20-30 fold speed-up for the Unisys PUNDIT natural language processing system, by exploring grammatical alternatives as or-parallel processes.
- Simulation technology. We designed and implemented a novel simulation system employing abstract interpretation that supports the simulation of concurrent behavior of a program inferred from its sequential behavior and a model of concurrency.

1.3. Juniper, a Logic + Functional Programming Language

Juniper is the name for a family of languages that combine the desirable features of logic programming languages and functional programming languages. The initial language, Juniper-A, provides a rudimentary logic programming capability, the pure Horn clause subset of PROLOG, available within the framework of SASL, a pure functional language. The key contribution of Juniper-A is the specification of the "bridging construct" that allows the functional component to invoke the logic component:

the logic expression is wrapped within a set notation and the values it represents are treated within SASL as a list. Juniper-A has been implemented with an interpretive implementation of the logic component in SASL.

Juniper-B1 is the first of a planned series of languages that provide a symmetric relationship between the logic and functional components. It provides most of the logic programming capabilities of PROLOG (excepting the "assert" and "retract" operations that alter the clause database and a few other non-logical features), again within a SASL framework. Juniper-B has proven to be a useful programming language, supporting significant portions of real applications. The ability to call functions from within logic clauses provides the programmer the appropriate choice of features, avoiding the typical PROLOG usage of defining a logical relationship when a simple function is desired, while providing the full capabilities of LP identified above. The symmetric relationship between logic and functions allows the programmer to move freely between them, subject only to the stricture that unbound logic variables have no meaning to the functional component, and will cause an error if encountered there.

Juniper-B1 has been implemented as an extension to a combinator-based SASL implementation. Five combinators have been added to allow compiling Juniper-B1 to combinators, and the logic inference mechanism implemented within the new combinator definitions. Section 2 describes Juniper-B1 and its implementation. We have also included as Appendix A a working paper which discusses the design issues and alternatives in more depth. (The working paper was written during the definition of Juniper-B1, and does not reflect the later evolution of the language and implementation.)

Later languages in the Juniper-B sequence, planned to include more advanced language features such as stronger typing and an assert/retract mechanism (elevating the clause database to an explicit data object), have not been defined.

The key characteristic of what we have termed Juniper-C languages is the integration of logic variables with the functional language. No Juniper-C language has been defined.

1.4. Compiling Logic Programs to Applicative Form

A major area of research has been the reduction of logic programs to a form that is compatible with the existing body of knowledge on support for functional programming languages. The key result in this area has been a rethinking of the role of logic variables and the unification process. Conventional LP systems treat logic variables as passive objects that, along with the other objects to be unified, are manipulated by a monolithic unification algorithm. In our new formulation, the objects to be unified are replaced by higher-order functions (called *gadgets*) that, applied to each other, accomplish the unification process. Thus, a gadget representing a variable and one representing a value will bind the variable to the value, which will appear wherever that instance of the variable is used in the program.

Implementation of gadgets requires a significant but carefully controlled violation of the normal rules for the lambda calculus, which underlies many functional language implementations. In practice, the only significant changes to functional language implementations that gadgets require are the introduction of specific allocation, value-fetching, and binding operations for gadgets, and restrictions on copying them. In particular, optimisation techniques for functional languages generally map into the extended model without serious problems. Section 3 describes the basic gadget technique and its application to Juniper-B1.

1.5. Gadget Technology applied to the Warren Abstract Machine

We have attempted to demonstrate the practicality of the gadget technology in several ways. First, a proof-of-concept system modified the existing Juniper-B1 implementation, stripping out the logic interpreter and replacing it with combinators to implement the primitive operations for gadgets (allocation of gadget instances, evaluation and binding of canonical representations, etc.). A source-to-source translator took Juniper-B1 code to pure SASL extended only with new "functions" to access these gadget mechanisms. The resulting system was able to run most features of Juniper-B1 (some uninteresting and problematical non-logical features were not re-implemented). Performance was in many cases better than the base implementation, mostly because the environment maintenance became much simpler.

With the gadget concepts demonstrated, we proceeded to ascertain the performance penalties one might pay for the optimisation potential they provide. We took the best publicly available implementation of PROLOG and modified it to use a gadget model of logic evaluation. The SB-Prolog implementation uses the Warren Abstract Machine (WAM) architecture as an intermediate code, with a byte-code emulator interpreting the instructions. We installed a set of gadget-oriented unification instructions, along with the necessary changes to the abstract architecture to produce what we call the "Gadgetised WAM" or GWAM, and modified the compiler to generate GWAM code. With remarkably little tuning, we found the the GWAM gave performance competitive with the WAM for most programs, edging ahead by 10-15% on some benchmarks. One problem with the current GWAM design gives the WAM a major advantage in the treatment of lists and structures as formal arguments; we have identified a remedy, but it requires more work on the compiler than the schedule allowed. Section 4 describes the Gadgetised WAM.

1.6. Partial Evaluation of Declarative Programs

One strength of gadget technology is that it supports powerful logic program optimisation techniques. The abstraction of logical variables, yielding a "functional" form of logical relations, allows standard functional language optimisation techniques to be applied to logic programs. Of these, abstraction and partial evaluation provide a single mechanism for a broad range of logic program optimisations. We have initiated work in this area, building necessary tools for partial evaluation and optimisation of functions; these tools will apply to logic relations in abstracted form without

major modification.

The major results in this area have been primarily in the partial evaluation at compilation time of functional expressions. The system is built on Turner's SASL reducer which is the basis for the Juniper-B1 sequential implementation. Function definitions are compiled to SKI combinator expressions, and expressions to be optimized are compiled and given to the compilation-time evaluator, with special "unevaluable" constants replacing unknown values. Reduction of the expression proceeds as long as progress is being made in simplifying, according to a fairly simple set of rules. For instance, a recursive function is expanded only once unless the simplification of the resulting expression removes the termination condition. Endless looping is avoided by putting an arbitrary limit on the amount of simplification. In practise, most expressions simplify quickly to an irreducible form; the amount of simplification varies substantially according to the expression. Appendix D describes the optimization techniques and shows some results.

1.7. Parallelism in the PUNDIT Parser

We have taken *or-parallelism* as a desirable form of parallelism to be exploited in parallel AI applications. In order to establish that this is a viable strategy, we have analysed and simulated the parallel behavior of an existing AI program component, the parsing module of the PUNDIT Natural Language Understanding system developed by Unisys. The simulations are driven by an application-specific model of parallel execution, and by the sequential behavior of the parser. The parallel behavior is based on simultaneous search across grammar rule disjunctions, with independent processes created for each disjunct when a disjunction is encountered. Since we desire all parses, no extra computation is performed over the sequential case; indeed, since backtracking is eliminated, some computation is eliminated. The introduced overheads for process creation and initialization are based on measured times, as are the durations of the steps of the computation, and reasonable assumptions about system overheads.

The results are quite encouraging: we found parallel speed-up factors of 10 to 20, with a minimum of 50% processor effectiveness, over a large range of sentences. The experimental methodology, which relies on the Virtual Computation Recorder (VCR) described below, supported the rapid simulation of performance over a range of system parameters, allowing the experiments to explore the trade-offs between processor effectiveness, overhead costs, and speed-up factors. The results indicate that the key characteristics of the application that allow the large speed-up factors are the size and shape of the search space, particularly the existence of long "garden paths" without a solution; the existence of multiple solutions and the requirement that all of them be found; and the separability of the search paths, allowing a simple model of parallelism without substantial synchronization. Section 5 describes our simulation study of the parallel behavior of the PUNDIT parser.

1.8. Juniper-B1 application programs

During the development of Juniper, we undertook to assess the utility of the language by translating into Juniper-B1 several existing logic programming benchmarks and substantial portions of two AI application programs (PUNDIT, a natural language understanding system, and KNET, a knowledge representation system, both developed at PRC). The timing of the effort helped dictate the methodology: to translate the existing Prolog programs to a "pure" subset of Prolog as we were defining Juniper, and then, when the Juniper definition and implementation were stable, to translate these pure Prolog programs to Juniper-B1. The latter translation was, in many cases, trivial, as the pure Prolog subset is also a subset of Juniper-B1. This translation was, however, in some cases to a more "idiomatic" use of Juniper-B1, using the power of both functional and logic programming elements of the language. The translations were made by several members of the PRC staff who were primarily Prolog programmers, some with strong Lisp backgrounds and some exposure to purely functional programming languages. None had any prior experience with lazy evaluation, which underlies the Juniper-B1 sequential implementation.

The pure Prolog translation removed usage of such inherently sequential (and non-logical) features as assertion and retraction of relations, and the Prolog *cut*, which, in a declarative language, is most closely approximated with an *if-then-else* construct. These programs are generally less efficient than the originals, since *cut*, in particular, is often used to control the flow of the sequential Prolog interpreter. We accepted this loss of efficiency, however, since the pure Prolog versions were only a step in the development of the Juniper versions.

The translation to Juniper-B1 varied from trivial (adjusting the syntax to fit the logic programming component of Juniper-B1) to substantial (e.g., rewriting relations used only as functions to be pure functions). Many of the translations, not surprisingly, took a great deal of effort. Several necessary or desirable new features of the language were identified in the process, and versions of the programs were produced as the features were defined and implemented. In addition, the essence of purely declarative program development and, in particular, debugging is distinctly unfamiliar to Prolog programmers. The insights gained in this effort for program development environments are described in Section 8.3.1. Appendix B describes the benchmark and small test programs that we translated, and describes some of the problematical features of the translation. Section 2-C includes the specification of the MicroPundit program.

1.9. Simulation of Concurrency by Abstract Interpretation

In the search for effective parallel execution models, we determined very early that conventional simulation techniques for parallelism were not sufficient to perform the sorts of experimentation that we needed to do. The richness of the possible models, coupled with the number of parameters to each, made it imperative that simulation runs be faster than detailed simulations, which typically run for hours or days on trivial programs.

The key to solving this problem was realizing that most of what happens in a program execution is uninteresting, from the parallelism point of view. By abstracting from the run only the interesting characteristics, but maintaining all the information that would drive a parallel execution of the program, we have been able to build a simulation system that allows us to run experiments quite rapidly on real application programs. This capability does not seem to have been achieved elsewhere, except in very specific domains.

The Virtual Computation Recorder, as the system is called, uses abstraction and abstract interpretation to provide a flexible simulation system that can be applied to a wide variety of concurrency analysis problems. It allows the experimenter to vary simulated system parameters easily, allowing the free exploration of the design space, without requiring the implementation of systems to be evaluated. This flexibility lets us ask "what if" questions about system architecture that would not otherwise be answerable. Section 6 describes the theory and capabilities of the VCR. Section 7 describes the VCR in somewhat more detail, including the details of its usage in the program.

1.10. Future Directions

Section 8 provides an overview of several research directions that flow naturally from the MASC program. We are actively pursuing some of these at this time; others will likely appear in future white papers and proposals. The major directions are (a) applying the compilation technology and parallelism analysis methods to new applications and languages, and (b) using the tools developed on the program in new areas.

Section 2

Definition of Juniper-B1: A Combined Logic + Functional Programming Language

by

Tom M. Blenko

Table of Contents

Section	Title	Page
2.1	Abstract	2-1
2.2	Introduction to Juniper	2-1
2.3	Juniper-B1: a simple functional/logic-programming language	2-4
2.3.1	Syntax of Juniper-B1	2-4
2.3.2	Operational semantics of Juniper-B1	2-12
2.4	Sequential implementation of Juniper-B1	2-20
2.4.1	Running Juniper-B1 under Unix	2-20
2.4.2	Deficiencies	2-21
2.4.3	Additional interpreter imperatives	2-22
2.5	Concurrent implementation of Juniper-B1	2-22
	References	2-24
Appendices		
2-A	Syntax of Juniper-B1	2-26
2-B	Juniper-B1 prelude functions	2-31
2-C	Juniper-B1 demonstration program (Micropundit)	2-37

Section 2

Definition of Juniper-B1: A Combined Logic + Functional Programming Language †

2.1. Abstract

Juniper is a programming language that unifies the two principal paradigms for Artificial Intelligence programming: functional programming, which is exemplified by pure LISP or SASL, and logic programming, whose best-known representative is PROLOG. Juniper makes available to the programmer a range of features from each paradigm: on the one hand, the abstract power of higher order functions, and on the other, the expressive power of multiple-solution relational programming and the logical variable. For the Juniper implementor, the existence of well-understood mathematical models for both language paradigms permits their unification in a single semantic model. The interpretation of this model is well-defined and, we anticipate, reducible to a set of primitive operations suitable for an efficient, distributed implementation.

This document contains a definition of the instance of Juniper currently under development, which we have named Juniper-B1. We first present a brief introduction to the principal issues guiding the development of Juniper and the relationship of Juniper to other proposed languages. There follow detailed definitions of the syntax and operational semantics of Juniper-B1. A companion document, *Issues in the Definition of Juniper* [1], discusses the language definition issues and their resolution in greater depth.

2.2. Introduction to Juniper

Juniper is a programming language based on the unification of the functional and logic-programming paradigms. The merging of these paradigms is important in two respects: first, the programmer is provided with a range of language constructs available from each paradigm, with the consequence that the most suitable ones may be applied to the programming task at hand; and second, the formal semantic underpinnings of the two language paradigms are well understood, and consequently it is possible to combine them into a unified semantic whole. This greatly simplifies the task of providing a single operational base for efficient, multiprocessor execution of Juniper programs.

For the programmer, Juniper can either be viewed as an extension of a logic-programming language (e.g. PROLOG) that includes reducible functions as terms, as found in first-order predicate logic, or as an extension of a lazy functional programming language (e.g., SASL) that includes a bag generator to "accumulate" solutions

† This section is also available as LBS Technical Report 78

generated by the logic-programming component. It is useful, then, to describe Juniper in terms of its functional-programming and logic-programming components, and the constructs which are used to combine them. Our goal, however, is to provide constructs that allow the programmer to switch freely from one paradigm to the other, so that Juniper can be viewed, for programming purposes, as a single, powerful, well-integrated programming language.

An important property of functional languages is *referential transparency* [20]. Referential transparency requires that the value of each instance of an expression, modulo variable renaming or changes of scope, be the same no matter where in a program it appears [15]. Another way of expressing this property is to say that a referentially transparent language is side-effect free, i.e., the evaluation of an expression does not affect, and is not affected by, the value of any other computation in the program. In particular, since we have a parallel execution environment in mind, it requires that the value of each expression be unaffected by the order in which other expressions are evaluated.

For example, consider the following recursive quicksort program in a pseudo-functional language:

```
define qsort(List):  
    if singleton(List)  
        return(List)  
    else  
        return(append(qsort(lesser_half(split(List,first(List)))),  
                       qsort(greater_half(split(List,first(List))))))
```

given

split(), which splits *List* into sublists of elements greater than and less than or equal to its second argument
lesser_half() and *greater_half()*, which select the lesser-than and greater-than sublists, respectively, returned by *split()*
append(), which appends two lists

There are a number of different, but still correct, execution sequences for *qsort* and its sub-processes (recursive calls). This is possible because the lack of interaction (via side-effects) among sub-processes allows considerable license in exploiting parallelism.

Note that a major characteristic of referentially transparent languages, and one quite different from imperative languages, is that they prohibit explicit assignment to variables. That is, variables in functional languages are created and bound only by function calls, so that variables have the *single-assignment property*. This greatly simplifies the multiprocessor implementation task, since a variable is bound as soon as it is created, and no inter-processor synchronization of a

(and indeed the ability to copy is sufficient access to remote variables).

Referential transparency is important because it results in reduced program complexity and because it greatly simplifies the implementation. These properties are reflected in claims of an order-of-magnitude greater code density for functional programs compared with imperative programs, and in the choice of functional languages as user languages for dataflow architectures [18]. Preservation of referential transparency is therefore an important constraint on the definition of Juniper.

Another important property of Juniper is its ability to be executed effectively on a highly-parallel computer architecture. In defining language constructs for Juniper, we are concerned with the opportunities for parallel execution that each construct provides. The constraint of referential transparency, however, disqualifies a major source of parallelism found in several of the proposed parallel (concurrent) logic-programming languages [4, 14, 19]: non-deterministic clause selection. For example, in Concurrent PROLOG, the clauses

```
shuffle([a|X]) :- | shuffle([X]).
shuffle([b|X]) :- | shuffle([X]).
```

(the "|" appearing in the clause bodies are guards signifying exclusive clause selection) would define a procedure *shuffle()* that, given a goal

```
?- shuffle([Z]).
```

would bind the logical variable Z to an arbitrary string of a's and b's. This is because clause selection is *non-deterministic*, which is to say left to the whim of the execution environment, and *exclusive*, i.e., exactly one of the clauses is selected. Juniper is less powerful than these other languages in this respect (and correspondingly easier to program with).

Since explicit concurrency constructs such as non-deterministic clause selection have not been included in Juniper, all parallelism is *implicit*. That is, concurrency is available by virtue of operations whose semantics permit parallel execution rather than appearing *explicitly* in the form of control constructs signifying concurrent task creation and management. Examples of operations in Juniper that support implicit parallelism are Boolean functions, logical OR-parallelism, and parallel evaluation of functional arguments.

Juniper provides *semantic* support for implicit concurrency in the form of what we have termed its *concurrent reduction model*. The concurrent reduction model guarantees that all active processes continue to get a share of the processing resource. This is often stated in a negative fashion: no active process is delayed indefinitely. So, for example, if the interpreter is to execute the Boolean AND operation, it might be the case that one conjunct will run forever, while another will return FALSE. In this case, we wish, in some finite amount of time, to run the conjunct producing FALSE to

completion and terminate the operation with the resulting value FALSE. Since it is not, in general, known which conjuncts will terminate and which will not, it is necessary to give all conjuncts a share of the processing resource so that they all may continue to proceed towards termination. In many existing programming languages, including SASL, for example, there is a sequential ordering on evaluation of conjuncts (the ordering may or may not be known to the programmer) that may result in non-termination of the conjunction as a whole (e.g., in the case described above).

This combination of implicit parallelism and concurrent reduction semantics simplifies the programmer's task substantially: programs can be written without regard for the underlying architecture or execution policies, and in particular without concern for the issues of safeness and liveness that must be addressed when explicit concurrency constructs are present.

The strategy for development of the Juniper language is to construct a sequence of languages that provide successively closer approximations to the eventual goal, a fully-integrated functional/logical programming language. The remainder of this document defines in detail the first full member in this sequence to be implemented, Juniper-B1.

2.3. Juniper-B1: a simple functional/logic-programming language

Juniper-B1 is defined in terms of

its functional-programming component;

its logic-programming component;

bridging constructs that allow integration of the two components;

a set of control policies that serve to define the operational semantics of the language as a whole, and especially the interaction of the two components via the bridging constructs.

In discussing the functional-programming and logic-programming aspects of Juniper-B1, we will rely on the reader's familiarity with the SASL functional programming language and the PROLOG logic-programming language, as discussed in Turner's SASL language manual [16] and in a standard reference for PROLOG [5] respectively.

2.3.1. Syntax of Juniper-B1

The syntax of Juniper-B1 is listed in BNF in Appendix 2-A. It reflects a commitment to backward compatibility with SASL, and with the CPROLOG, DEC-10, and Quintus implementations of PROLOG [3,10,11]. This serves two purposes at the present time: first, it permits programmers with prior experience using functional languages or PROLOG to get started writing Juniper-B1 programs quite readily; and

second, it simplifies the task of translating SASL and PROLOG application programs to Juniper-B1, which is one of the initial routes to developing Juniper-B1 applications.

Syntax of the Juniper-B1 functional component

The grammar productions in Appendix 2-A down to and including that defining the *<filter>* non-terminal are obtained directly from the SASL syntax, with the following qualifications:

1. The *<logic_expr>* non-terminal in the production for *<simple>*

```

<simple>      ::=      <logic_expr>
<logic_expr> ::=      { < > | <goal_list> }
                |      { < <logic_var_list> > | <goal_list> }

```

represents the bridging construct that permits the functional component to invoke the logic-programming component. It will be discussed in greater detail in the section Syntax of Juniper-B1 bridging constructs below.

2. The *<logic_var>* terminal in the same production

```

<simple>      ::=      <logic_var>

```

represents logical variables used as terms; these are used as part of the bridging construct that permits the logic-programming component to call the functional component.

3. All strings will be delimited by double-quote characters. This is compatible with implementations of PROLOG and Burroughs Austin Research Center SASL [12], but not with Turner's SASL implementation [16].
4. Strings will be treated as syntactic sugar for lists of character constants. This is consistent with their treatment in CPROLOG and DEC-10 PROLOG, and their treatment in SASL.
5. "[]" and "NIL" are equivalent to SASL's "()", denoting the empty list.
6. The "^" character is the infix Boolean OR operator, rather than the "|" character as in SASL.
7. The SASL functions *logical()* and *not()* have been renamed *boolean()* and *negate()* respectively.

8. The *<program>* non-terminal is not, strictly speaking, part of SASL or Juniper-B1. These productions are specific to the Turner implementation of SASL and the sequential implementation of Juniper-B1: they allow queries to the interpreter, and declaration of function definitions and logic clauses.

The "offside" convention in SASL requires sub-expressions continued on more than one line to appear to the right of the parent expression on the page. This is something of a nuisance, but serves syntactically as a terminator on function definitions and some other expressions. Juniper-B1 preserves the "offside" convention.

The reserved identifiers in Juniper-B1 are to some degree implementation-specific; the list included in Appendix 2-A is derived from the sequential implementation of Juniper-B1. Overlap between function names and predicate names is prohibited.

In other respects, Juniper-B1 functional definitions and expressions are entirely compatible with SASL programs (as accepted by the Turner interpreter), and SASL programs will be directly executable by the Juniper-B1 interpreter.

Syntax of the Juniper-B1 logic-programming component

Syntax of the logic-programming component of Juniper-B1 differs in several important respects from the syntax of extant PROLOG implementations. The change is intended to reinforce syntactically certain semantic features of Juniper-B1 that are different from those found in PROLOG. These differences are introduced in the remainder of this document. Generally speaking, the "pure" subsets of PROLOG and the Juniper-B1 logic-programming component, which is to say the subsets lacking *assert()*, *retract()*, *cut()*, *not()*, *functor()*, *is()*, and other "meta" operators, are syntactically identical. It is in the introduction of "meta" operations that syntactic and semantic differences appear.

The syntax of the logic-programming component appears in Appendix 2-A, beginning with the *<logic_expr>* production and continuing to the end of the grammar. This syntax is highly compatible with that of the "pure" subset of PROLOG. Logical variables are signified by capitalised names, and constructors by uncapitalized names in the usual way. A single underscore ("*_*") used as a term is interpreted as an anonymous variable.

In addition, these predicates are included in the Juniper-B1 logic-programming component:

Group 1:

numberp()	succeeds if its argument evaluates to a numeric value
charp()	succeeds if its argument evaluates to a character value
booleanp()	succeeds if its argument evaluates to a Boolean value
listp()	succeeds if its argument evaluates to a list structure
name()	succeeds if the name of the first argument (an identifier) consists of the alphanumeric constants listed in the second argument.

Group 2:

var()	succeeds if its argument is an unbound variable
non_var()	returns negation of var()
bound()	returns same as non_var()
unbound()	returns same as var()
head()	succeeds if its second argument is bound to the head of its first argument
tail()	succeeds if its second argument is bound to the tail of its first argument
==() (prefix form)	succeeds if both args can be shown equal without use of unification
\==() (prefix form)	returns negation of ==()
=() (explicit unify operator, prefix form)	succeeds if two arguments are unifiable

Group 3:

call()	succeeds if its argument can be proven as a goal.
fail()	returns negation of call().
if-then-else()	succeeds if first and second arguments can be proven; else if first argument cannot be proven, but the third argument can.
or()	is an inclusive or, as in PROLOG, and succeeds with all solutions from both arguments.

Note that all operators except for the *Group 1* predicates are definable in terms of *call()* and *fail()* in Juniper-B1 - see the Issues document [1] for discussion - so they are not defined as primitives in Juniper-B1. Operators in the first two groups have the conventional PROLOG (prefix) syntax, except as noted. Operators in the last group (*call()*, etc.) are discussed below in the sections Definition of *call()* and *fail()* meta-operators and Definition of the *if-then-else()* meta-operator.

Several PROLOG operators are not included in Juniper-B1. Five of these will very likely be included in a future version of Juniper:

=..() pronounced *univ*, decomposes a constructor term into a list containing its principal functor and its argument terms.

functor(), arg() like **=..()**, are selector operations on constructions. They are equivalent in power to **=..()**. Both the logic-programming and functional components can currently perform selection on constructors as part of clause matching; however, more powerful selection operations may be included in subsequent versions of Juniper.

bagof(), setof() can be defined, in forms weaker than their PROLOG counterparts, using the bridging constructs. The full implementations require more powerful constructor-decomposition operations than those found in Juniper-B1.

The remainder will probably not be included in future revisions of Juniper, but substitutes have been or will be included:

is() must have as its second argument an arithmetic expression, which is evaluated and bound to its first argument. This operation is achieved in Juniper-B1 by using the arithmetic expression as a functional term and forcing its evaluation and subsequent binding via the bridging constructs and unification (see the section *Function-call bridging construct* below).

assert(), retract() are altogether absent from Juniper-B1; substitutes for *assert()* and *retract()*, probably in the form of explicit user-definable databases, will be introduced in a future version of the Juniper language.

read(), write() Juniper-B1 currently permits I/O only via the primitive facilities inherited from SASL (i.e., the result of an evaluation is printed out). All I/O from the logic-programming component must be effected using these operations via the bridging constructs. A more sophisticated approach to incorporating I/O is a desideratum for a future version of the Juniper language.

op() *op()* is a PROLOG directive for specifying operator precedence that is very useful in practice. Its inclusion is an issue that will be considered as part of the larger issue of providing a more sensible, unified syntax for future versions of Juniper.

Additional "meta" operators from PROLOG are generally definable using the operators listed above.

Definition of call() and fail() meta-operators

The Juniper-B1 *call()* operator has the same syntactic and semantic definition as the PROLOG *call()* operator. *call()* takes a formula as an argument, and succeeds (with associated variable bindings) if the formula is derivable from the logic-programming database. *call()* is a primitive operation in Juniper-B1 that is implemented using bridging constructs in both directions to reinvoke the logic-programming component (in a future version of Juniper with more powerful selection on constructors, *call()* and *fail()* will not be primitive, but will be defined directly in terms of the bridging constructs).

fail() behaves like PROLOG's *not()*. Its success/failure semantics are the opposite of *call()*'s, and it never returns bindings. The purpose of renaming *not()* to *fail()* is to leave it to the programmer to define what "not" should mean: it might mean negation-by-failure, which is what Juniper-B1's *fail()* and PROLOG's *not()* implement, or it might mean something else that the programmer may choose. *fail()* is also a primitive operation in Juniper-B1, with an implementation similar to *call()*'s.

Definition of the if-then-else() meta-operator

The Juniper-B1 *if-then-else()* operator has the familiar

$P \rightarrow Q ; R$

syntax from PROLOG. The interpretation is that if *P* succeeds, then return the result of attempting to prove *Q* in the environment produced by *P*; otherwise if *P* fails, return the environment resulting from the attempt to prove *R*. *if-then-else()* is defined directly in terms of *call()* and *fail()*.

In conjunction with the *=()* (unify) operator, *if-then-else()* can serve as a substitute for PROLOG's *cut()* operator, although its behavior somewhat different, and more in line with the OR-parallel character of Juniper-B1. A typical use of PROLOG's *cut* is in a sequence of clauses

```
p(<arg11>,<arg12>) :- <cond1>, !, <action1>.
p(<arg21>,<arg22>) :- <cond2>, !, <action2>.
.
.
.
p(<argN1>,<argN2>) :- <actionN>.
```

The scope of the *cut()* is the entire set of clauses for *p()*, which are understood to be searched sequentially from top to bottom. This can be expressed in an almost equivalent form using the Juniper *if-then-else()* construct, syntactically $(P \rightarrow Q);R$, in the following way:

```

p(A,B) :- (=[A,B],[<arg11>,<arg12>]), <cond1> -> <action1> ;
          (=[A,B],[<arg21>,<arg22>]), <cond2> -> <action2> ;
          .
          .
          .
          (=[A,B],[<argN1>,<argN2>])) -> <actionN> ;
fail.

```

where $=()$ is the predicate designating unifiability.

The principal respect in which *cut()* differs from this use of *if-then-else()* is that, if the "cut" $\langle cond \rangle$ succeeds, only the first solution found for the associated $\langle cond \rangle$ will be used to satisfy the $\langle action \rangle$ goals. With PROLOG's *cut()*, determining which solution is "first" depends upon the ordering of the clause database. In Juniper-B1, it is in general indeterminate what the first solution satisfying a $\langle cond \rangle$ is, since the database is not ordered, so there is no corresponding determinate restriction available. Consequently, every solution for the Juniper-B1 $\langle cond \rangle$ is used to generate solutions for the succeeding $\langle action \rangle$ goals in the "then" branch. It should be noted that Juniper-B1 is, as a consequence, much more faithful to the general objective of limiting proceduralism and maintaining completeness with respect to the logical interpretation of programs.

Syntax of Juniper-B1 bridging constructs

The Juniper-B1 bridging constructs are unique to the Juniper language, and so we will examine their syntax carefully. The bridging construct that enables the functional component to invoke the logical component is termed a *bag generator*, and is derived from Turner's work on relative set generators [17], and that of Darlington [6] and Robinson [13] on absolute set generators. Our construct is termed a bag generator because an element in the bag may appear more than once. The bridging construct enabling the logic-programming component to invoke the functional component can be thought of as an extension to PROLOG that permits reducible first-order function applications to be used as logical terms, provided that any logical variables they contain are grounded prior to reduction time.

Syntax of bag-generator bridging construct

The semantics of the bag-generator bridging construct dictates that it produce a lazy list of bindings satisfying a particular goal solved by the logic-programming component. Syntactically, it has the following form

```
{ < <template> > | <goal> }
```

in which $\langle template \rangle$ is a tuple of logical variables that appears inside a pair of $\langle goal \rangle$ is a logic formula that constitutes a goal for the logic-programming

component. *<goal>* will normally contain instances of the variables found in the template: these represent the universal variables to be solved for by the logic-programming component. The following is an example of the use of the bag-generator bridging construct:

```
{ <Name,Date_of_hire> | employee(Name,_,Date_of_hire,_) }
```

This expression will return a list of all name/date-of-hire pairs from the database (as in PROLOG, the "_" term is treated as an anonymous variable).

Note, however, that the sense in which the template identifiers should be considered logical variables is limited. Although this is discussed in more detail in the section **Bridging constructs and the logic-programming component** section below, we outline some of the relevant information here:

Template identifiers are treated as local variables and their scope is bounded by the bag-generator expression.

The goal in the bag-generator may contain no (unbound) variables except those listed in the template.

The only effect of bag-generator evaluation is to return a bag of solutions. Each solution in the list represents a set of simultaneous bindings to the local variables that satisfies *<goal>*. There is no exporting of bindings via logical variables.

Since unbound logical variables may be returned in solutions, all variables returned will be "named apart" from any other variable present in the program. Since logical variables have no meaning in the functional-programming component, the return of variables in solutions is meaningful only when the bag-generator has been invoked as part of the recursive call of the logic-programming component from itself.

Syntax of function-call bridging construct

The function-call bridging construct is the means by which the logic-programming component invokes the functional component. It requires no change in the syntax of the logic-programming component. The operational change is achieved by treating *some* logic-programming terms as reducible function calls. Both the Juniper-B1 data structures and **Bridging constructs and the logic-programming component** sections below should be consulted for additional discussion of the semantics.

Note that by using the traditional syntax inherited from PROLOG terms we introduce dual syntax for function calls. In particular, the expression

$(f\ a\ b)$

in the functional component signifies the function f applied to the arguments a and b , while the expression

$f(a,b)$

is the equivalent form on the logic-programming component. Unifying the two forms is a goal of future revisions to the syntax.

Juniper-B1 data structures

Juniper-B1 inherits its syntactic structures from both SASL and PROLOG. SASL permits two types of structures: list structures and function applications. PROLOG includes user-defined constructors, which appear as principal functors in PROLOG terms, and list structures. Juniper-B1 incorporates all of these: it overlays SASL and PROLOG list structures, includes (reducible) function applications from SASL, and permits arbitrary irreducible constructors as found in PROLOG.

For list structures, the functional component inherits its syntax directly from SASL, and the logic-programming component inherits its syntax directly from PROLOG. The two are not the same, but are unambiguous in context.

Function applications and user-defined constructors are syntactically indistinguishable. In the functional-programming component, both inherit the syntax of SASL function applications. In the logic-programming component, both inherit the PROLOG syntax for user-defined constructors. The distinction between reducible function applications and irreducible user-defined constructors is made at run time, depending upon whether a function definition has or has not been declared for the identifier in the function position. We will have more to say about this in the Functional semantics of Juniper-B1 section below.

2.3.2. Operational semantics of Juniper-B1

By operational semantics of Juniper-B1 we mean the operations the interpreter performs in order to execute a particular program. Our description is intended to give the programmer a tangible sense of what an interpreter will actually *do* when a Juniper-B1 program is executed.

We will rely, once again, on the programmer's familiarity with SASL and PROLOG, so that the discussion mainly reflects the ways in which Juniper-B1 differs from SASL and PROLOG, rather than providing an exhaustive rehearsal of all the operations included. The reader is directed to the companion document [1], for a more in-depth discussion of the motivation for, and issues pertaining to, the definition of Juniper-B1 semantics.

Functional semantics of Juniper-B1

The semantics of functional reduction for Juniper-B1 is very close to that of SASL. The areas we need to elaborate on are the differences between sequential and concurrent reduction semantics, the issues relating to the bridging constructs between the functional and logic-programming components, and the treatment of user-defined constructors.

Concurrent reduction model and termination properties

One major advantage of assuming a concurrent reduction model is that it changes termination properties, relative to the serial reduction model, in a way that simplifies the programmer's task. In particular, if one is computing the Boolean AND of a set of conjuncts, it is necessary to exercise caution in certain cases: one of the conjunct evaluations may result in a non-terminating computation, and this in spite of the fact that another of the conjuncts evaluates to FALSE. In a serial reduction model, it is necessary for the programmer to anticipate this situation, and assure that a potentially-FALSE conjunct is evaluated prior to any conjunct which might fail to terminate. Otherwise, the computation, and indeed the whole program, will fail to terminate.

In Juniper-B1 and its successors we are committed to concurrent reduction model semantics. This means that computational resources are shared in such a way that non-termination never occurs in the case cited above: each conjunct gets fair access to the processor resource, and eventually the failing conjunct itself terminates and forces a non-local exit of the entire computation. The principal operations affected are Boolean AND and OR functions, and the bag-generator bridging construct, which returns results of OR-parallel logic-programming computations in the order that they become available (this is discussed further in Functional semantics of bag-generator bridging constructs below).

It is also worth noting that by providing a concurrent semantics with a fairness policy for what is essentially a lazy functional language, we bring the semantics of the lazy language much closer to that of a call-by-value language. That is, if we had an eager implementation that respected fairness as discussed above, the language semantics supported by the implementation would be very close to that defined for Juniper-B1, which is lazy.

We feel that this semantic model does a more satisfactory job of capturing the computational model a programmer would like to have, i.e., it saves the programmer from being concerned about termination issues whenever this is possible.

Function-call bridging construct

The functional reducer accepts no ungrounded logical variables as arguments to functions. A logic-programming expression must assure that any logical variables which appear in a program as arguments to a function are fully grounded (contain no embedded logical variables) at reduction time, as stipulated in the Juniper-B1

unification/reduction policy section below. Any violation of this policy will result in an error from the run-time system -- the violation is interpreted as an illegal program, rather than as an inability to reduce.

For example, if the interpreter is attempting to satisfy a goal

$p(1+X,Z)$

using a clause

$p(2,A) :- p2(A).$

the variable X must be bound to an integer in the calling environment, so that the functional operation (addition) may be performed as a precondition for matching the goal's first argument to the first argument of the clause head. This amounts to testing for equality with the number 2 as part of unification. Otherwise the run-time system will report an error. The complete rules for interaction of unification and reduction are discussed below in the section Juniper-B1 unification/reduction policy.

Note that exactly the same rule applies to bag generators as to other functional expressions. That is, if the bag generator is invoked from the logic-programming component, the *<goal>* part of the bag-generator expression may contain variables that are free in the bag generator, i.e., are not included in the *<template>* variable list. The bound, or local, variables are exactly those that appear in the *<template>*. As with other functional expressions, the free variables must be grounded at reduction time for the generator.

Functional semantics of bag-generator bridging constructs

A major element of Juniper-B1's concurrent reduction semantics, and a fundamental property of the bag generator that acts as a bridging construct, is that an enumeration of the elements returned by the bag generator reflects the order in which the corresponding OR-parallel logic-programming computations terminate. The resulting termination properties differ from those normally found in sequential reduction models, since if the sequential model begins a non-terminating computation, the entire program will fail to terminate. This ordering property permits non-terminating OR-parallel computations to be deferred indefinitely, while others receive a share of the computational resource and continue to be returned as they terminate.

This policy raises a pertinent issue with respect to referential transparency. Bags generated by the bridging construct are themselves unordered, but it is generally necessary to apply an enumeration function to bags in order to access their elements. It is the enumeration order that is determined by the implementation, according to our concurrent reduction semantics. But this results in a violation of referential transparency if different instances of the same bag expression are present, and if the implementation assigns different enumeration orderings (it cannot, in general, know that

another identical instance has already been computed). For example, the expression

`one_of ({<X>|p(X)})`

might return the "first" solution to $p(X)$, but it may not return the same value at each place in the program where it is invoked, since the run-time system may produce different enumeration orders under different circumstances.

We had two major options in defining Juniper-B1. The first, modelled after more conventional sequential programming languages, was to guarantee *a priori* a particular enumeration order on bags, and leave to the programmer the task of assuring that no non-terminating computation in the enumeration ordering would render inaccessible the results of computations appearing later in the ordering. The second, which we have chosen, was to relieve the programmer of the problem of ordering the computations by exploiting the parallel execution environment. Issues related to this choice are discussed in more detail in the Issues document [1].

The Juniper-B1 policy addressing this issue is as follows:

Each bag generator instance is treated as if it is unique. That is, under the terms of referential transparency, each instance of a bag generator at run-time will be treated as if it is different from every other bag-generator instance. The programmer, then, can determine whether two bag generator instances are equal only by comparing them element-by-element, but has no reason to expect that the enumeration order of two otherwise identical bags will be the same.

If we had a special function such that each time it was called at run-time it returned a different value, say a clock function, we could effect this policy by including the function in each and every bag generator, thus forcing instances of the bag generators to be unique.

We instead achieve the same result invisibly. One may wish to view the policy as a violation of referential transparency, which it is, strictly speaking. However, we find that it does not reintroduce the kinds of problems we have sought to avoid with referential transparency: specifically, the side effects that force serialization or synchronization upon us. Therefore it is, in a tangible sense, a "safe" violation of referential transparency.

Constructors and normal forms in Juniper-B1

One way in which Juniper-B1 extends SASL is in the addition of user-defined constructors. Constructors are treated as function applications in which the function name has no associated function definition, and is therefore irreducible. From one point of view, this amounts to an extension to the set of normal forms conventionally defined for a functional language. That is, certain "values" for expressions are being

admitted which formerly would have been treated as errors. There is more to it than that, however. SASL forbids equality tests between functions or expressions used as functions. Our treatment of constructors consists not only of admitting the irreducible forms, but of extending the equality relation to permit comparison of irreducible forms as well. So it is advisable to view user-defined constructors as having the same footing as SASL list structures, rather than as special forms of function application.

The inclusion of additional structures in the functional component, with a corresponding extension of equality, is reflected, as well, in the "pattern-matching" capability for clause heads. If, for example, one had an irreducible constructor *record()*, and wished to define a function *f* which operates only on *record()*'s, a straightforward approach in Juniper-B1 would be something like

DEF

f (record *a b c*) = <body of definition using values bound to *a*, *b*, and *c*>

?

Note that the constructor name and arity must match exactly. A more powerful matching capability, which is directly dependent upon the power of Juniper's selection operations on constructors, is a desideratum for future versions of Juniper.

The motivation for including irreducible constructor terms is simple: it provides the utility of PROLOG terms, which is desired, and in particular permits the definition of Skolem functions. Logically speaking, Skolem functions permit one to proceed with logical deduction as if one knew what a particular term designates without actually knowing the definition of the function (for the purpose of reduction). Interestingly enough, the inclusion of constructors also is very much in the spirit of current work on introducing data types into functional languages [7-9, 18].

Logic-programming semantics of Juniper-B1

The semantics of logic programs is the area in which Juniper-B1 most differs from its SASL and PROLOG predecessor. Most of the changes are motivated by the desire to exploit available parallelism. The argument might also be made that, from a logic-programming point of view, the resulting semantics is simpler and more natural, as we have argued for the concurrent reduction model in the functional-programming component (see Concurrent reduction model and termination properties above). But in part this remains an open question.

We have divided the discussion into two parts: the first addresses issues that relate only to the logic-programming component, and the second examines the issues that arise in consequence of the bridging constructs.

Semantics of Juniper-B1 as a logic-interpreter

The chief difference between interpretation of PROLOG and interpretation of Juniper-B1 is in the search strategy: PROLOG incorporates left-to-right depth-first search, while Juniper-B1 incorporates left-to-right breadth-first search. This is analogous to the distinction between serial and concurrent reduction models discussed above for functional reduction semantics. It has a pronounced, and we believe desirable, effect on the termination properties of the search: since the search remains breadth-first, no subtree indefinitely delays search of alternate subtrees. That is to say, a non-terminating subtree can never prevent other subtrees from terminating and returning their results.

In more concrete terms, this means that the entire database is searched in parallel in order to satisfy a goal, and each clause head that matches the goal activates an independent "process" to generate one or more binding sets. This is known as an OR-parallel reduction model because all eligible clauses are activated and may proceed concurrently.

It should therefore be evident why *cut()* had to be eliminated in favor of *if-then-else()*: *cut()* requires a sequentialization of clauses in the database. And while some degree of sequentialization is necessary in order to achieve *if-then-else()*, the global scope of a single *cut()* with respect to a procedure name in the database was unnecessarily restrictive.

A minor difference between Juniper-B1 and PROLOG is in the naming of the negation-by-failure operator. Juniper-B1 names the operator *fail()*, leaving the programmer with the option of defining *not()* in terms of *fail()*, or according to some other criterion.

Bridging constructs and the logic-programming component

Definition of the bridging construct that allows the logic-programming component to invoke the functional component is probably the most difficult part of the Juniper definition task. A central issue is that the functional component does not admit logical variables, so that terms used in a call to the functional programming component must be fully grounded. This not only restricts the use of reducible terms in the logic-programming component, but it forces upon the programmer the task of assuring that the terms will in fact be fully grounded at the point during execution where the reduction takes place. This restriction runs counter to the goal of providing a programming language which is logical, and hence non-procedural, in character.

Operationally, control over the reduction of functional terms is an integral part of the unification procedure, so the interaction between unification and reduction is the area we define first.

Juniper-B1 unification/reduction policy

Since reducible function applications can appear as terms in logic programs, it is necessary to have a policy governing the reduction of these applications as part of

unification. The Juniper-B1 unification/reduction policy is about the simplest that suffices, and may be thought of as a form of lazy unification:

1. No higher-order functions are permitted as terms in the logic-programming component.
2. All reduction is lazy. That is, reduction will only proceed as far as required in order to satisfy (or fail) the equality test required for unification. This is consistent with the reduction semantics of SASL and the functional component of Juniper-B1.

In particular, a variable may be unified to a function application, even though the application is not yet reduced, and perhaps may be not reducible, as yet, because it is not fully grounded.

3. Equality is restricted to normal forms. That is, according to the extended normal forms defined earlier for constructors in the functional component, two terms are equal for the purposes of unification if and only if their normal forms have been computed and are structurally identical. This allows us to by-pass the task of trying to satisfy unifiability by comparing function applications prior to, or intermediate forms in the course of, reduction, which is very expensive to implement.

4. In the case that no ordering on binding of variables within the clause head permits the unification to proceed, the unification fails. Otherwise the unification succeeds. Note that since Juniper-B1 unification has the Church-Rosser property, every ordering which permits unification to succeed produces the same unifier.

For example, if the function f is reducible but the variable X is not grounded, then the following unification cannot proceed:

$$\text{unify}([f(X), 1], [Y, Y])$$

That is, Y can be bound to $f(X)$ and Y can be bound to 1 , but it is not possible to test for the equality of $f(X)$ and 1 because X is not grounded, and $f(X)$ cannot be reduced.

From a programming point of view, we have extended the unification algorithm familiar from PROLOG:

Unification in Juniper-B1 proceeds as unification in PROLOG, except that

1. *Groundedness condition:* Any reducible (functional) sub-term *must* become grounded in the course of unification if an equality test is to be performed upon it. Otherwise the program is illegal.
2. *Ordering condition:* If there is any ordering of variable binding during unification of the two terms that permits satisfaction of the groundedness condition, then the unification will proceed.

This policy in Juniper-B1 is experimental, and subject to revision as part of the ongoing Juniper development process. The present policy has been chosen for its simplicity, with the idea that simplicity is highly desirable from a programming point of view.

Unification and the Church-Rosser property

Recall that ungrounded functional terms (those containing ungrounded logical variables) may not be reduced, and any attempt to do so produces a run-time error. Consequently, the programmer must be certain that in all cases a logical variable used as an argument to a function is bound prior to reduction time. The implementation of Juniper-B1 needs to assure that if there is any ordering on sub-term unification that will result in a variable's being bound at reduction time, then the implementation will find and apply it.

PROLOG unification, and more generally first-order unification, has the Church-Rosser property: that is, the unifying substitution obtained is the same irrespective of the order in which sub-terms are unified. Juniper-B1 unification also has the Church-Rosser property modulo functional reduction, e.g., a unifier with $f(1)$ as the value for a variable X is indistinguishable from an otherwise equivalent unifier with the value 6 for X , assuming that $f(1)$ equals 6 . It should be noted that this "indistinguishability" equivalence relation among unifiers does not change the termination properties of the unification algorithm. The programmer need only be sure that there exists one ordering under which logical variables prior to reduction, and the implementation need only be able to find one such ordering, in order for the implementation to successfully obtain the unifier intended by the programmer. The Juniper-B1 unification algorithm and its properties are presented in considerable detail in a companion document [2].

Coincident with this assurance we need to declare the *scope* of the unification: might it apply only to the term used as an argument to a predicate, or might it, for example, apply across the whole of the unification operation matching a goal to the head of a clause? In Juniper-B1, we have opted for the latter, which is to adopt the widest syntactic scope within which unification occurs. Note that this policy has a particular implication with respect to our substitute of *if-then-else()* for *cut()*. The use of the *if-then-else()* operation in the body of a clause in place of *cut()* means that all the clause-head matching operations that occur in a system with *cut()* must, in Juniper-B1, be performed explicitly in the body of the clause. For example, the PRO-

```
p(X,1) :- p1(X),!,q1(X).
p(X,2) :- p2(X),!,q2(X).
```

would be rewritten in Juniper as

```
p(X,Y) :- (=(Y,1),p1(X)) -> q1(X) ;
          (=(Y,2),p2(X)) -> q2(X) ;
          fail.
```

But if the matching in the head of $p(X,Y)$ were substantially more complex, the rewritten form in Juniper would still have to be expressed with a single *unify()* goal in each condition of the body in order to capture the effect of the original PROLOG code, given the limitations of the scope policy for unification that has been defined.

2.4. Sequential implementation of Juniper-B1

This section of the document describes release 1.10 of the Juniper-B1 sequential implementation. The companion document, *Issues in the Definition of Juniper (working paper)* [1], and the SASL Reference Manual [16], provide much of the context for the following discussion, and are presumed as prerequisites to the information presented here.

The chief distinction between the *sequential* and *concurrent* implementations of Juniper-B1 is that the former does not include concurrent reduction semantics (of which fair evaluation of conjuncts and breadth-first search in the logic-programming component are important examples).

As a functional language interpreter, the Juniper-B1 interpreter provides additional functionality in comparison to the current (Turner's) implementation of SASL, with respect to reporting of syntax errors, and with respect to debugging.

2.4.1. Running Juniper-B1 under Unix

The sequential Juniper-B1 interpreter can be found in the `~blenko/juniperb1` directory on the following PRC Unix systems: `sol` and `testbed`. Also in that directory are copies of the source files listed in appendices B and C, and a subdirectory containing complete sources and the Makefile (in RCS format) for the current release.

The Juniper-B1 interpreter is invoked in the same fashion as the SASL interpreter. In response to a shell command line prompt, type "juniperb1" in order to enter the interpreter. Except as documented below, the execution environment is exactly the same as that found in Turner's SASL interpreter. The initial invocation sequence should look something like this:

Juniper-B1 interpreter, Release: 1.10, Date: 5/1/87
implemented as an extension of
Turner's SASL interpreter, Revision: 1.5, Date: 02/18/86
there will now be a short delay while juniperb1 gets its act together...
hello from juniperb1(say OFF to quit)
Juniper-B1>

The user responds by typing an interpreter directive or a Juniper-B1 expression to be evaluated. After each expression evaluation, but not after interpreter directives, the interpreter will prompt with

what now?

In addition, whenever the interpreter is expecting further input, after a line-feed in the middle of a function definition, for example, it will prompt with

Juniper-B1>

The appendices have been included to help those getting started with this implementation. Appendix 2-B contains the prelude functions that are loaded automatically on each invocation of the Juniper-B1 interpreter. Appendix 2-C contains several short programs used to debug the current implementation. These should provide a starting place for the novice Juniper-B1 programmer.

Please inform Tom Blenko (blenko@PRC.unisys.com) of any errors in the implementation or this document.

2.4.2. Deficiencies

Specific deficiencies of the current implementation with respect to a correct and complete implementation of sequential Juniper-B1 include the following:

The predicates `==()`, `\==()`, and `name()` are not implemented at this time, and we are not aware of their being a high priority.

The "unifies" predicate, conventionally signified with concrete syntax `=(arg1,arg2)`, instead appears as `equal()` for implementation-dependent reasons.

The logic-programming `or()` operator has not been implemented.

While there are no known deficiencies in the implemented unification algorithm with respect to its specification (as found in this document), this is a possible source of errors.

2.4.3. Additional interpreter imperatives

Several imperatives have been added to extend the underlying Turner SASL interpreter on which this implementation is based. Chief among these are the following directives:

DEFS – causes all functions and the associated combinator expressions which stand as their definitions to be printed out. The order of the functions is the order in which they are defined. Note that this includes the "prelude" functions, which are user-level functions that are automatically loaded every time that the interpreter is invoked.

DECS – causes all clause definitions to be printed out. The order is chosen by the implementation, but all clauses with the same functor and arity are grouped together.

TRACE *<function_name>* – enables a trace when an expression with *<function_name>* initially in its function position is reduced. Each expression, as it is printed out, is annotated with the numeric level of recursive call to the reducer in which the expression is being reduced. Note that the effect of combinator reduction is to generate a sequence of rewritings of the combinator expression corresponding to *<function name>*'s definition, so that tracing a function virtually assures a plethora of combinator expression output.

UNTRACE *<function name>* – undoes the effects of **TRACE**.

TRACER – prints out all combinator expressions as they are reduced. Output is so abundant that it will overwhelm all but the hardest (or foolhardest) souls.

DUMP – enables the **TRACER** flag and, for each expression printed out, does a complete dump of all active memory locations at the time the expression is reduced.

SHELL – implements an escape, with no side effects, to the shell specified in the **SHELL** environment variable.

2.5. Concurrent implementation of Juniper-B1

Work has begun on a concurrent implementation of Juniper-B1. A description of the implementation and its usage will be inserted here as that information becomes available.

Acknowledgements

The many contributions of Lynette Hirschman, Bill Hopkins, and Don McKay have been an essential part of the research effort reported in this document.

References

1. Blenko, T. M., *Issues in the Definition of Juniper (working paper)*, Logic-based Systems Group, Paoli Research Center, Unisys Corp., 1987.
2. Blenko, T. M., *Juniper-B1 Unification (working paper)*, Logic-based Systems Group, Paoli Research Center, Unisys Corp., 1987.
3. Bowen, D. L., Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D., *Decsystem-10 Prolog User's Manual*, Occasional Paper 27, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, November, 1982.
4. Clark, K. L. and Gregory, S., *PARLOG: A Parallel Logic Programming Language*, Research report DOC 84/4 (revised), Imperial College of Science and Technology, London, June, 1985.
5. Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, Springer-Verlag, New York, 1981.
6. Darlington, J., Field, A. J. and Pull, H., *The unification of functional and logic languages*, Research report DOC 85/3, Department of Computing, Imperial College of Science and Technology, 1985.
7. Harper, R., MacQueen, D. and Milner, R., *Standard ML*, ECS-LFCS-86-2, University of Edinburgh. Department of Computer Science. Laboratory for Foundations of Computer Science, 1986.
8. Milner, R., *A Theory of Type Polymorphism in Programming*, *Journal of Computer and System Sciences* 17, 3 (1978), 348-375.
9. Milner, R., *A proposal for Standard ML*, *Proc. of 1984 ACM Conf. on Lisp and Functional Programming*, Austin, Texas, 1984, 184-197.
10. *C-Prolog User's Manual*, Version 1.5, September 8, 1984.
11. *Quintus Prolog User's Guide*, version 6, Quintus Computer Systems Inc., April 1986.
12. Richards, H., *An Overview of ARC SASL*, in *SIGPLAN Notices Notices*, vol. 18, October, 1984.
13. Robinson, J. A., *New Generation Knowledge Processing/Syracuse University Parallel Expression Reduction: First annual progress report*, Syracuse University, December, 1984.
14. Shapiro, E. Y., *A subset of Concurrent Prolog and its interpreter*, Tech. Rep. 003 (revised February 1983), ICOT - Institute for New Generation Computer Technology, Tokyo, Japan, January, 1983.
15. Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge Massachusetts, 1977.
16. Turner, D. A., *SASL Language Manual*, University of St. Andrews, 1976.
17. Turner, D. A., *Functional Programming and its Applications - An Advanced Course*, Cambridge University Press, 1982.

18. Turner, D. A., Miranda: A non-strict functional language with polymorphic types, *Lecture Notes in Computer Science*, , 1985, 1-16.
19. Ueda, K., Guarded Horn Clauses, Tech. Rep. 103, ICOT – Institute for New Generation Computer Technology, Tokyo, 1985.
20. Whitehead, A. N. and Russell, B., *Principia Mathematica*, Cambridge University Press, 1913.

Appendix 2-A: Syntax of Juniper-B1

Functional programming component:

In the following definition, angle braces ($\langle \rangle$) are used as meta-brackets, * is used to signify zero or more occurrences of a subexpression, and + is used to signify one or more occurrences of a subexpression. The lone angle brackets in the initial portion of the $\langle \text{logic\_expr} \rangle$ productions (before the vertical bar) are part of the language syntax.

$\langle \text{program} \rangle$	::=	$\langle \text{expr} \rangle ?$ DEF $\langle \text{defs} \rangle ?$ DEC $\langle \text{logic\_clauses} \rangle ?$
$\langle \text{expr} \rangle$	::=	$\langle \text{expr} \rangle$ WHERE $\langle \text{defs} \rangle$ $\langle \text{condexp} \rangle$
$\langle \text{condexp} \rangle$	::=	$\langle \text{opexp} \rangle \rightarrow \langle \text{condexp} \rangle ; \langle \text{condexp} \rangle$ $\langle \text{listexp} \rangle$
$\langle \text{listexp} \rangle$	::=	$\langle \langle \text{opexp} \rangle , \rangle^* \langle \text{opexp} \rangle$ $\langle \text{opexp} \rangle ,$
$\langle \text{opexp} \rangle$	::=	$\langle \text{prefix} \rangle \langle \text{opexp} \rangle$ $\langle \text{opexp} \rangle \langle \text{infix} \rangle \langle \text{opexp} \rangle$ $\langle \text{comb} \rangle$
$\langle \text{comb} \rangle$	::=	$\langle \text{comb} \rangle \langle \text{simple} \rangle$ $\langle \text{simple} \rangle$
$\langle \text{simple} \rangle$	::=	$\langle \text{name} \rangle$ $\langle \text{constant} \rangle$ ($\langle \text{expr} \rangle$) $\langle \text{xfexpr} \rangle$ $\langle \text{string} \rangle$ $\langle \text{logic\_expr} \rangle$ $\langle \text{logic\_var} \rangle$
$\langle \text{defs} \rangle$	::=	$\langle \text{defs} \rangle ; \langle \text{clause} \rangle$ $\langle \text{clause} \rangle$
$\langle \text{clause} \rangle$	::=	$\langle \text{namelist} \rangle = \langle \text{expr} \rangle$ $\langle \text{name} \rangle \langle \text{rhs} \rangle$
$\langle \text{rhs} \rangle$	::=	$\langle \text{formal} \rangle \langle \text{rhs} \rangle$ $\langle \text{formal} \rangle = \langle \text{expr} \rangle$
$\langle \text{namelist} \rangle$	::=	$\langle \langle \text{struct} \rangle , \rangle^* \langle \text{struct} \rangle$ $\langle \text{struct} \rangle ,$
$\langle \text{struct} \rangle$	::=	$\langle \text{formal} \rangle : \langle \text{struct} \rangle$ $\langle \text{formal} \rangle$
$\langle \text{formal} \rangle$	::=	$\langle \text{name} \rangle$ $\langle \text{constant} \rangle$

		(<namelist>)
		(<appl>)
		<string>
<appl>	::=	<name> <formal>
		<appl> <formal>
<constant>	::=	<numeral>
		<charconst>
		<boolconst>
		0
		[]
		<string>
<numeral>	::=	<real> <scale-factor>
		- <real> <scale-factor>
<real>	::=	<digit>+
		<digit>* . <digit>+
<scale-factor>	::=	e <digit>+
		e - <digit>+
<boolconst>	::=	TRUE
		FALSE
<charconst>	::=	% <any char>
		SP
		NL
		NP
		TAB
		NIL
<string>	::=	"<any message not containing a double-quote>"
<sfexpr>	::=	{ <expr> ; <qualifiers> }
		{ <qualifiers> }
<qualifiers>	::=	<generator>
		<qualifiers> ; <generator>
		<qualifiers> ; <filter>
<generator>	::=	<name> <- <expr>
		<name> <- <generator>
<filter>	::=	<expr>

Logic-programming component:

<logic_expr>	::=	{ <◇> <goal_list> }
		{ < <logic_var_list> > <goal_list> }
<body>	::=	<goal_list> -> <goal_list> ; <body>
		<goal_list>

<goal_list>	::=	<goal>
		<goal> , <goal_list>
<goal>	::=	formula
		(<body>)
<formula>	::=	<name>
		<name> (<term_list>)
<term_list>	::=	<term> <, <term_list> >*
<term>	::=	<op_term>
		<list_term>
		<name> (<term_list>)
<op_term>	::=	<prefix> <op_term>
		<op_term> <infix> <op_term>
		<simple>
<list_term>	::=	[]
		[<term_list>]
		[<term> <term>]
		<string>
<logic_var_list>	::=	<logic_var> <, <logic_var> >*
<logic_clauses>	::=	<logic_clause> +
<logic_clause>	::=	<formula> :- <body> .
		<formula> :- .
		<formula> .

Operators (in order of increasing binding power)

This is a corrected list of the operators provided in Turner's SAS^T manual [16].

:	++	-	infix (right associative)
..			infix (non-associative)
...			postfix
^			infix (left associative)
&			infix (left associative)
-			prefix
>>	>	>=, =, ~=, \=, <=, <, <<	infix (non-associative)
+	-		infix (left associative)
+	-		prefix
*	/	DIV, REM	infix (left associative)
**			infix (right associative)
#			prefix
.			infix (right associative)

Additional syntactic conventions

/* ... */	serves as a comment delimiter
	reserves remainder of input line for comments
lower case identifiers	serve as names and functional variables (e.g. <name>)
capitalized identifiers	serve as logical variables (e.g. <logic_var>)

The following upper case identifiers are reserved:

COUNT	INTERACTIVE	SMART
DEC	K	SP
DECS	NIL	TAB
DEF	NL	TO
DEFS	NP	TRACE
DIV	OBJECT	TRACER
DUMP	OFF	TRUE
FALSE	READ	UNTRACE
GC	REM	WHERE
GET	RESET	WRITE
I	S	

The following lower case identifiers are defined by the system in the sequential implementation, and may be changed at the user's peril:

abs	fail1	numberp
all	filter	or
and	foldl	pi
append	foldr	plus
arctan	for	p_q_r
boolean	from	printwidth
booleanp	function	product
bound	hd	reverse
call	head	rjustify
call1	interleave	show
char	intersection	sin
charp	iterate	some
cjustify	lay	spaces
code	layn	sqr
concat	length	sum
cons	letter	tail
cos	list	take
count	listdiff	times
decode	listp	tl
digit	ljustify	true
digitval	log	unbound
drop	map	unify
e	member	union
entier	mkset	until
eq	name	var
equal	negate	while
exp	non_var	zip
fail	number	

Appendix 2-B: Juniper-B1 prelude functions

```
DEF || Juniper-B1 prelude file
    || Last revised May 1987
```

```
abs x = x<0 -> -x ; x
```

```
all = foldr and TRUE
```

```
and x y = x&y
```

```
append x y = x++y
```

```
||arctan      inverse trig function - primitive to Juniper-B1
```

```
||boolean     type testing function - primitive to Juniper-B1
```

```
||code ch = integer code for ch in local character set - primitive to Juniper-B1
```

```
cons a x = a:x
```

```
||char        type testing function - primitive to Juniper-B1
```

```
cjustify fieldwidth x
```

```
    =spaces lmargin,x,spaces rmargin
```

```
    WHERE
```

```
    margin = fieldwidth - printwidth x
```

```
    lmargin = margin DIV 2
```

```
    rmargin = margin - lmargin
```

```
concat = foldr append ()
```

```
||cos         trig function - primitive to Juniper-B1
```

```
count a b =      a>b -> ()
              a:count(a+1)b
```

```
||decode n    the character whose integer code is n - primitive to Juniper-B1
```

```
digit x = code x >= code %0 & code x <= code %0
```

```
digitval x = code x - code %0
```

```
drop 0 x = x
```

```
drop n() = ()
```

```
drop n (a:x) = drop(n-1)x
```

e = exp 1

||entier primitive to Juniper-B1

eq x y = x=y

||exp exponential function - primitive to Juniper-B1

filter f () = ()

filter f(a:x) = f a -> a:filter f x ; filter f x

foldl op r () = r

foldl op r(a:x) = foldl op(op a r)x

foldr op r

 =f

 WHERE

 f () = r

 f(a:x) = op a(f x)

for a b f = a>b -> () ; f a: for(a+1)b f

from n= n:from(n+1)

||function type testing function - primitive to Juniper-B1

hd(a:x)= a

I x = x

interleave = concat.zip

intersection x y = filter(member y)x

iterate f x = x:iterate f(f x)

K x y = x

lay () = ()

lay (a:x) = show a:NL:lay x

layn x = f 1 x

 WHERE

 f n () = ()

 f n(a:x) = n:") ":show a:NL:f(n+1) x

```
length() = 0
length(a:x) = 1+length x
```

```
letter x = code x >= code %a & code x <= code %z ^
          code x >= code %A & code x <= code %Z
```

```
||list      type testing function - primitive to Juniper-B1
```

```
listdiff () y = ()  || listdiff defines the action of the "-" operator
listdiff x () = x
listdiff(a:x)(b:y) = a=b -> listdiff x y
                  listdiff(a:listdiff x(b,))y
```

```
ljustify fieldwidth x
  = x ,spaces(fieldwidth - printwidth x)
```

```
||log      natural logarithm - primitive to Juniper-B1
```

```
map f () = ()
map f (a:x) = f a: map f x
```

```
member x a = some(map(eq a)x)
```

```
mkset () = ()
mkset (a:x) = a:filter(negate.eq a)(mkset x)
```

```
negate x = ~x
```

```
||number   type testing function - primitive to Juniper-B1
```

```
or x y = x ^ y
```

```
pi = 4*arctan 1
```

```
plus x y = x+y
```

```
printwidth () = 0
printwidth(a:x) = printwidth a + printwidth x
printwidth TRUE = 4
printwidth FALSE = 5
printwidth x
  = number x -> nwidth x; 1
  WHERE
    nwidth x
      = x<0 -> nwidth(-x)+1
```

```

    abs(x-entier(x+0.5))<2e-6 & abs x<=1e7 -> intwidth x
    .001<=x & x<=1000-> intwidth x + 7
    12
intwidth x
    = x<10-> 1
    1 + intwidth(x DIV 10)

```

product = foldr times 1

reverse = foldl cons ()

rjustify fieldwidth x = spaces(fieldwidth - printwidth x), x

```

show x  = x==() ^ ~list x ^ haschars 6 x -> show1 x
        show1(hd x):(tl x ==()->%,;showlis(tl x))
        WHERE
        show1 () = "()"
        show1 NL = "NL"
        show1 NP = "NP"
        show1 TAB = "TAB"
        show1 x
            = char x -> %%,x
            list x
            ->   haschars 6 x -> %":showstr x
                %(:show1(hd x):(tl x==()->%,;showlis(tl x)):"")
            x
        haschars 0 x = TRUE
        haschars n () = TRUE
        haschars n(a:x) = char a & haschars (n-1) x
        showlis () = ()
        showlis(a:x) = %,;show1 a:showlis x
        showlis x = ")++":show1 x
        showstr ()= %"
        showstr x  =   char(hd x)-> hd x:showstr(tl x)
                    %":")++":show1 x

```

sin trig function - primitive to Juniper-B1

some = foldr or FALSE

spaces = map(K %).count 1

sqrt primitive to Juniper-B1


```

take 0 x = ()
take n () = ()
take n (a:x) = a:take(n-1)x

times x y = x*y

tl(a:x) = x

union x y = filter(negate.member y)x ++ y

until f g x = while(negate.f)g x

while f g x
  = f x-> while f g(g x)
    x

zip x    =  hd x->()->()
           map hd x:zip(map tl x)

```

? || End of Juniper-B1 functions definitions

```

/*
 *   Logic clause definitions
 */

```

DEC

```

booleanp(X) :-
    equal(boolean(X),TRUE).

```

```

bound(X) :-
    non_var(X).

```

```

call(P) :-
    call1([P],P).
call1([Solution|Rest],Solution).
call1([Forget|Rest],Solution) :-
    call1(Rest,Solution).

```

```

charp(X) :-
    equal(char(X),TRUE).

```

```

fail(P) :-
    fail1([P | I]).
fail1([]).

head([Head|Tail],Head).

p_q_r(P,Q,R) :-
    prove_all(P), prove_all(Q).
p_q_r(P,Q,R) :-
    fail(prove_all(P)), prove_all(R).

prove_all([X|Rest]) :-
    call(X), prove_all(Rest).
prove_all([]).

listp([X|Y]).

non_var(X) :-
    var(X) -> fail; true.

numberp(X) :-
    equal(number(X),TRUE).

tail([Head|Tail],Tail).

true.

unbound(X) :-
    var(X).

unify(X,X).

var(X) :-
    fail(fail(equal(X,1))), fail(fail(equal(X,2))).
?
```

Appendix 2-C: Juniper-B1 demonstration program (Micropundit)

This appendix contains the principal files of the Juniper-B1 version of the *Micropundit* natural language system. The lexicon files have been omitted for the sake of brevity. We first present an example of a query to Micropundit.

The test program *demo1* is a query to parse a sentence:

```
demo1 = NL,%T,he,SP,disk,SP,is,SP,failing,%.,NL:
        (hd {<Tree> | parse([the,disk,is,failing,%.],Tree)})
```

It produces this output:

The disk is failing.

```
—
sentence
  center
    assertion
      sa
        null
      subject
        lnr
          ln
            tpos
              t == dict(the,[[def]])
            apos
              null
            npos
              null
            n == dict(disk,[[singular,11]])
          rn
            null
        sa
          null
      ltr
        lv
          null
        tv == dict(is,[[12,singular,present]])
      rv
        null
    sa
      null
  object
```

```

be_aux
  vingo
    lvingr
      lv
        null
        ving == dict(failing,[[12]])
      rv
        null
    sa
      null
      object
      nullobj
    sa
      null
  sa
    null
  . == dict(.,[feature(root,.),punct])

```

```

/*
 *   interpreter.j1
 *
 *   This file contains the Restriction Grammar interpreter that
 *   is used by microPundit. There are some small differences
 *   between this interpreter and the full Restriction Grammar
 *   interpreter used by Pundit.
 */

```

DEC

```

|| execute(+Label,-Tree,+InWords,-OutWords)
||
|| execute/4 takes a nonterminal name and a list of word definitions, and
|| returns a parse tree for that nonterminal and any remaining words.
execute(X,Tree,InWords,OutWords):-
    exec(X,Tree,_,%,InWords,OutWords,[]).

||
||
|| exec(+RuleBody,+FirstNode,-LastNode,+ParentNode,+InWords,-OutWords,+Anc)
||
||
|| exec/7 is the main recursive relation in the interpreter. It has
|| clauses for each of the cases that can appear in a grammar rule.
|| As a side effect of interpreting the grammar rule, it constructs
|| a parse tree. This is done using the FirstNode and LastNode
|| parameters. FirstNode is the position in the tree to build the
|| first node constructed under RuleBody, while LastNode is the position
|| to build the next node after RuleBody. ParentNode is used by
|| restrictions (not in the current implementation). Anc is the list
|| of rules in a path from the current node to the root of the
|| tree. This is used (if needed) to detect left-recursion (not
|| needed for the current grammar).
||
||
|| conjunction
exec(comma(X,Y),First,Last,R,InWords,OutWords,Anc):-
    exec(X,First,Next,R,InWords,IOtmp,Anc),
    exec(Y,Next,Last,R,IOtmp,OutWords,Anc).

|| disjunction
exec(semi_colon(X,Y),First,Last,R,InWords,OutWords,Anc):-
    exec(X,First,Last,R,InWords,OutWords,Anc).
exec(semi_colon(X,Y),First,Last,R,InWords,OutWords,Anc):-

```

```

exec(Y,First,Last,R,InWords,OutWords,Anc).

|| restrictions (currently just succeed).
exec(restriction(X),Prev,Prev,R,IO,IO,Anc).

|| terminals
exec(kleene(X),Here,Next,_,InWords,OutWords,Anc):-
    label(Here,X),
    attach_word(X,Here,InWords,OutWords),
    sibling(Here,Next).

|| literals
exec([Hd|Tl],First,Last,R,InWords,OutWords,Anc):-
    ewords([Hd|Tl],First,Last,R,InWords,OutWords).

|| the empty string
exec(%~,Prev,Prev,_,IO,IO,Anc).

|| nonterminals
exec(X,First,Next,_,InWords,OutWords,Anc):-
    execPred(X,First,InWords,OutWords,Anc),
    sibling(First,Next).

|| execPred(+Label,+Parent,+InWords,-OutWords,+Anc)
||
|| execPred/5 is used to expand nonterminals. It looks up the
|| grammar rule (or rules) for a nonterminal using rewrites_to,
|| and then calls exec/7 with the rulebody for the nonterminal.
execPred(X,Parent,InWords,OutWords,Anc):-
    rewrites_to(X,Y),
    label(Parent,X),
    daughter(Parent,YTree),
    exec(Y,YTree,_,Parent,InWords,OutWords,Anc).

|| ewords(+Literal,+Here,-Next,+Parent,+InWords,-OutWords)
||
|| ewords/6 is used to expand a list of literals.
|| Currently, only literals one token long are allowed.
ewords([Hd],Here,Next,_,InWords,OutWords):-
    attach_literal(Hd,Here,InWords,OutWords),
    sibling(Here,Next).

|| attach_literal(+Literal,+Node,+InWords,-OutWords)

```

```

|| attach_literal/4 is used to attach a single token literal to
|| the parse tree.
||
attach_literal(Literal,Node,[dict(Literal,Defns)|OutWords],OutWords):-
    label(Node,Literal),
    getWdField(Node,dict(Literal,Defns)).

|| attach_word(+Label,+Tree,+InWords,-OutWords)
||
|| attach_word/4 is used to attach terminal symbols to the parse tree.
|| It uses the function find/2 to make sure that the next word
|| is of the appropriate lexical class, and then attaches the
|| relevant lexical information to the parse tree.
||
attach_word(X,Node,[dict(Word,Defn)|OutWords],OutWords):-
    equal(ClassDefn,find(X,Defn)),
    fail(equal(ClassDefn,FALSE)),
    getWdField(Node,dict(Word,ClassDefn)).

```

?

```

/*
 *   path.j1
 *
 *   This file contains all the tree building/traversing primitives.
 *   Most of the routines contained in this file are not used in the
 *   current implementation, but are included for the sake of compatibility
 *   with Pundit, and will be used eventually when restrictions are
 *   added to microPundit.
 */

```

DEC

```

/*
 *   Node constructors. There are two node constructors, daughter/2
 *   and sibling/2. These do not actually build tree structures,
 *   just build node paths, which may later become instantiated into
 *   tree structures.
 */

```

```

|| daughter(+Node,-Daughter)
||

```

```

|| daughter/2 constructs the path for the node that will be the daughter of
|| Node. It does not actually add that node to the parse tree (This
|| occurs whenever Daughter become instantiated to a tree term.
daughter(link(tt(L,Daughter,Sib,W,Sem),Path),
          link(Daughter,up(tt(L,_,Sib,Wd,Sem),Path))).

```

```

|| sibling(+Node,-Sibling)
||

```

```

|| sibling/2 constructs the path for the node that will be the sibling of
|| Node. Like daughter/2, it does not actually add that node to the
|| parse tree.
sibling(link(tt(L,D,Right,Wd,Sem),Path),
         link(Right,left(tt(L,D,_,Wd,Sem),Path))).

```

```

/*
 *   Tree Traversal. There are four operations for traversing trees:
 *   down/2, up/2, left/2, and right/2. These operations do not
 *   build structure, and will fail if attempting to travel to a
 *   nonexistent node.
 */

```



```

down(link(SSS,InP),link(E,up(SSS,InP))) :-
    equal(SSS,tt(,_E,_,_)),
    non_var(E).

```

```

right(link(SSS,InP),link(E,left(SSS,InP))) :-
    equal(SSS,tt(,_E,_,_)),
    non_var(E).

```

```

up(link(SSS,up(P,R)),link(P,R)) :-
    nonvar(P),
    equal(P,tt(,_SSS,_,_)).
up(link(,_left(X,Rest)),Out) :-
    nonvar(X),
    up(link(X,Rest),Out).

```

```

left(link(SSS,left(E,OutP)),link(E,OutP)):-
    nonvar(E),
    equal(E,tt(,_SSS,_,_)).

```

```

/*
 *   Other Selector Functions -- These functions are used to select
 *   certain pieces of information from the node.
 */

```

```

root(link(tt(,__,_Word,_,_),Root) :-
    var(Word) ->
        equal(Root,Word) ;
    equal(Word,dict(,_,[feature(root,Root)]_)).

```

```

sem_info(link(tt(,__,_Sem,_,_),Sem) :-
    non_var(Sem).

```

```

attrlst(link(tt(,__,_dict(,_AL,_,_),_,AL) :-
    non_var(AL).

```

```

getatts(link(tt(,__,_dict(W,Atts,_,_),_,Atts) :-
    non_var(W).

```

```

varattr(link(tt(,__,_dict(,_Att,_,_),_,Att) :-
    var(Att).

```

```

checkWdField(link(tt(____,W,_,_),_),Flag) :-
    non_var(W),
    equal(W,Flag).

```

```

/*
 *   Constructors – These functions are generally used as constructors,
 *   although they could be used as selectors.
 */

```

```

treeTerm(tt(____,_,_)).

```

```

label(link(tt(L,____,_,_),_),L).

```

```

nullPath(TreeTerm,link(TreeTerm,%~)).

```

```

word(link(tt(____,dict(W,_,_),_),_),W).

```

```

wordDef(link(tt(____,dict(W,Def,_,_),_),_),dict(W,Def)).

```

```

putattr(link(tt(____,dict(____,A,_,_),_),_),A).

```

```

getWdField(link(tt(____,W,_,_),_),W).

```

```

removeSibs(link(tt(L,C,____,W,Sem,_,_),_),tt(L,C,____,W,Sem)).

```

```

colon_args(dict(Wd,Atts),Wd,Atts).

```

```

colon_args(dict(Wd,____,Atts),Wd,Atts).

```

```

|| findAtt
|| searches for the first argument of the functor "feature" in 2nd arg list
|| if first arg has no functor "feature", then search for 1st arg in 2nd
|| arg list
||
findAtt(A,AttrList) :-
    member_p(A,AttrList).
findAtt(feature(T,____),AttrList) :-
    member_p(T,AttrList).

```

```

make_list(Type,List) :-
    isa_list(Type) -> equal(List,Type) ; equal(List,[Type]).

```

```

isa_list([]).
isa_list([_]).

```

```

member_p(A,[A!_]).
member_p(A,[_!List]):-
    member_p(A,List).

```

?

DEF

```

|| find takes two arguments, a terminal symbol, and a word definition list.
|| it returns FALSE if no element of the list is of the form a or a:Def,
|| otherwise it returns the empty list if it found an item of the form a,
|| or it returns Def if it found an item of the form a:Def.

```

```

find a () = FALSE
find a (x:y) = (a = x) -> () ; find_helper1 a (univ_call x) y
find a b = (a = b) -> () ; find_helper2 a (univ_call b)

```

```

find_helper1 a b y = a = (hd (tl b)) -> tl (tl b) ; find a y

```

```

find_helper2 a b = a = (hd (tl b)) -> tl (tl b); FALSE

```

```

univ_call x = (hd (hd { <Univ> | univ_db(x,Univ)}))

```

?

```

/*
 *   print_tree.j1
 *
 *   This file contains the pretty printer for trees in the RG style.
 *   It depends on the file pp.j1 (the pretty printer for sets and
 *   logic items).
 */

```

DEF

```

tab 0 = ()
tab n = SP: tab (n-1)

```

```

display_contents = pp

```

?

DEC

```

print_tree(link(Tree,Path),Output) :-
    print_tree_term(Tree,0,Output,[NL]).

```

```

print_tree_term(Tree,Indent,Output,Output) :-
    var(Tree).

```

```

print_tree_term(Tree,Indent,[NL|append(tab(Indent),[Label|Output])],Tail) :-
    non_var(Tree),
    equal(Tree,tt(Label,Child,Sib,Word,Sem)),
    print_word_field(Word,Output,Temp1),
    print_tree_term(Child,Indent + 3, Temp1,Temp2),
    print_tree_term(Sib, Indent,Temp2,Tail).

```

```

print_word_field(Word,In,In) :-
    var(Word).

```

```

print_word_field(Word,
    append([SP,%=,%=,SP,display_contents(Word)],Tail),Tail) :-
    non_var(Word).

```

```

|| These clauses for univ_db help the pretty printer for logic terms to
|| know how to take apart terms.
univ_db(link(TT,Path),[link,TT,Path]).
univ_db(tt(Label,Child,Sib,Word,Sem),[tt,Label,Child,Sib,Word,Sem]).
univ_db(dict(Word,Def),[dict,Word,Def]).
univ_db(feature(Label,Value),[feature,Label,Value]).

```

univ_db(idiom(First,Rest),[idiom,First,Rest]).

?

Section 3

**Compiling Logic Programs
to Applicative Form**

by

Tom M. Blenko

Table of Contents

Section	Title	Page
3.1	Introduction	3-1
3.2	Definition of gadgets	3-2
3.2.1	Simple variable and constant gadgets	3-3
3.2.2	Structure gadgets	3-12
3.2.3	Non-functional operations in gadget definitions	3-21
3.2.4	Correctness, parallelizability, and evaluation policies	3-21
3.2.5	Additional perspectives on gadgets	3-22
3.3	Implementation via combinator reduction	3-23
3.3.1	Implementing gadgets via combinators	3-24
3.3.2	Gadget definitions	3-24
3.3.3	Gadget code	3-25
3.4	Gadgets and optimizing transformations	3-29
3.4.1	Permissiveness of gadget code	3-30
3.4.2	Logic-programming-specific optimizing transformations	3-30
3.5	Conclusion	3-37
	References	3-38

Section 3

Compiling Logic Programs to Applicative Form

3.1. Introduction

We have recently been engaged in an effort to implement Juniper, a functional+logical programming language, using combinator reduction in a parallel execution environment. Techniques for translating functional programs to combinator expressions are well known [7, 12, 23], but the problem of translating logic programs to combinator expressions is as general as translating them to functional programs, and remains open. This paper describes a technique for doing this translation based on higher-order functions, termed *gadgets*¹, and demonstrates its applicability in the context of a combinator-based implementation.

The Juniper language (or class of languages) is based on OR-parallelism, where OR-parallelism is extended in the functional context to include fair evaluation of disjuncts and conjuncts in Boolean expressions. For the present discussion, it suffices to view Juniper as a pure, OR-parallel logic-programming language to which *call()* and *not()* have been added, but not *assert()* and *retract()*. An *if-then-else()* construct is defined in terms of *call()* and *not()* and serves as a substitute for the infamous PROLOG *cut()*.

The translation of such a language to functional form requires resolution of these four sub-problems:

- Translation of logic-programming control constructs to functional programming/ λ calculus.
- Mapping of unification control behavior to the λ calculus.
- Mapping of logical variable binding and scope to the λ calculus.
- Support for logical variables in the context of multiple solutions.

In previous work, Carlsson [3] and Nilsson [20] implemented logic-programming control constructs in a functional interpreter using either proof (environment) streams or continuations, and Haynes [10] has explored the latter approach in considerable detail. The gadget technique relies heavily on continuation passing, so an encoding of control constructs based on continuations was the natural choice for our combinator-based implementation. An implementation based on proof streams is feasible as well, and perhaps preferable for a performance-oriented single-processor implementation.

¹*gadget*st, n. Any article for which a name cannot be readily found or thought of at the moment.

Carlsson and Nilsson implement logical variables as *cons* structures, pass variable-binding environments explicitly as arguments, and use an explicit unification routine in their interpreters. Finn [6], Robinson and Greene [8, 22], and Bage and Lindstrom [1] take a more sophisticated approach in their combinator-based interpreters for lazy functional languages with logical variables: the unification operation appears explicitly at the combinator level, and functional terms that stand in for logical variables are overwritten with their values at binding time. Consequently, there are no variable-binding environments, *per se*. Haynes [10] and Felleisen [5] have taken a similar approach, using Scheme as the target language.

The gadget technique separates the variable-binding operations from the treatment of the control behavior subsumed by unification. The *BIND* operation appears as a primitive in section 2, which defines the gadgets that are the heart of our translation technique. In section 3, which describes our combinator-based implementation of one member of the Juniper family using gadgets, *BIND* is implemented as an overwriting operation, in the fashion of Finn, Robinson and Greene, Bage and Lindstrom, Haynes, and Felleisen. Other implementations of *BIND* are possible, moreover, and the choice of implementation is largely independent of the gadget technique itself.

The treatment of unification control behavior is the heart of the gadget technique, and is addressed in depth in section 2. It permits elimination of explicit unification in all cases by modeling logical variables not as values, but as higher-order functions termed *gadgets*. At an intuitive level, these functions encode the operational behavior of logical variables described separately in proposals by Lindstrom [15] and Wise [25].

In support of multiple solutions, the Finn and Felleisen implementations provide depth-first search with "undoing" of variable bindings on backtracking; Robinson and Greene provide breadth-first search via copying of unbound logical variables on spawning. Bage and Lindstrom's language does not provide for multiple solutions, and Haynes' treatment demonstrates how to achieve a variety of search strategies. The gadget technique has no direct bearing on the issue of control strategies and their implementation.

Other benefits of gadgets may arise from their use as an intermediate language for the compilation of logic programs. In section 4 we provide simple examples demonstrating how powerful optimizations can be applied to logic programs after translation to gadget code. Transformations that eliminate logical variables may also be enlisted in pursuit of the broader goal of translating logic programs to purely functional languages.

3.2. Definition of gadgets

The intuition behind gadgets is simple: given any two terms, we want a functional encoding for each such that when one is applied to the other, the effect on the program is identical to that achieved by unifying the corresponding terms. So, for example, given two terms t_1 and t_2 and their encodings, signified by $\langle t_1 \rangle$ and $\langle t_2 \rangle$, the

$$(<t_1> <t_2>)$$

is to return failure when t_1 and t_2 are non-unifiable, to have no effect if t_1 and t_2 are identical, and so forth. Furthermore, it must be the case that for all t_1 and t_2 ,

$$(<t_1> <t_2>) = (<t_2> <t_1>)$$

In essence, we are requiring the unification operation to be directly translated to β -reduction, the principal operation underlying functional reduction.

The encodings that achieve this are what we have termed *gadgets*. Their definition is based on modeling unification as a "negotiated" settlement between two parties (the terms), in which a simple or complex term is always willing to signal and relinquish control to its partner. A variable term, on the other hand, will seize control irrespective of its partner and irrespective of whether the partner has previously relinquished control. Each gadget, then, consists of a negotiating algorithm that assures the desired result irrespective of the gadget to which it is applied. As we demonstrate in what follows, this model is readily implemented by a simple coroutining procedure based on continuation passing.

We use the notation of the λ calculus throughout the definitions that follow. While certainly the addition of non-functional operations mean that the language is no longer the λ calculus, the notation does serve a useful purpose, as we shall see, since for certain purposes, e.g., program transformation, it may nevertheless be treated as the λ calculus

We will also assume throughout that the evaluation policy is lazy, or leftmost first. The role of laziness in the correctness of gadget definitions will be addressed in the section below entitled, *Correctness, parallelisability, and evaluation policies*.

3.2.1. Simple variable and constant gadgets

Initial definitions for constant and variable gadgets are given by

```
( $\lambda N.\lambda A.$ 
  if ( $A == \%$ )
    then ( $\lambda V.$ if ( $V == 1$ )
      then  $N$ 
      else fail)
    else ( $A\ N\ \% 1$ ))
```

representing the constant 1, and

```

(λN.λZ.
  if (Z==%)
    then if (Xc==UNBOUND)
      then (λB.(BIND Xc
        (λN1.λA.
          if (A==%)
            then (λV.if (V==B)
              then N1
              else fail)
            else (A N1 % B))
          N))
    else (Xc N %)
  else if (Z==<X>)
    then N
  else if (Xc==UNBOUND)
    then (BIND Xc Z N)
  else (Xc N Z))

```

representing the variable X . We write $\langle 1 \rangle$ and $\langle X \rangle$ as shorthand names for these gadgets.

Before examining the structure of these gadgets in detail, we provide a simple example of their intended use. Suppose we wish to unify X and 1, with the result (the binding of X to 1) to have effect in the remainder of the program, signified by a continuation *cont*. This would be expressed by the function application

$(\langle X \rangle \text{ cont } \langle 1 \rangle)$

(application associates to the left) which is reduced via the following sequence of β reductions (some intermediate steps are elided, $[v/z]t$ signifies the term t with the value v substituted for all free occurrences of z , and $\%$ signifies a special constant). Substi-

tuting first for the gadget $\langle X \rangle$, the expression becomes

```
((λN.λZ.
  if (Z==%)
    then if (Xc==UNBOUND)
      then (λB.(BIND Xc
        (λN1.λA.
          if (A==%)
            then (λV.if (V==B)
              then N1
              else fail)
            else (A N1 % B))
          N))
    else (Xc N %)
  else if (Z==⟨X⟩)
    then N
  else if (Xc==UNBOUND)
    then (BIND Xc Z N)
  else (Xc N Z))
cont
<1>)
```

and then

```
(if (<1>==%)
  then if (Xc==UNBOUND)
    then (λB.(BIND Xc
      (λN1.λA.
        if (A==%)
          then (λV.if (V==B)
            then N1
            else fail)
          else (A N1 % B))
      cont))
  else (Xc cont %)
else if (<1>==⟨X⟩)
  then cont
else if (Xc==UNBOUND)
  then (BIND Xc <1> cont)
else (Xc cont <1>))
```

Under an assumption we will not justify at present, that the variable X_c appearing here has been previously substituted for by the constant *UNBOUND*, the reduction

continues with

(BIND Xc <1> cont)

and with the *BIND* expression read as "Return the continuation *cont* with all free occurrences of *Xc* replaced by <1>", the final result is

([<1>/Xc]cont)

which is the desired effect of the unification operation.

Structure of simple gadgets

We now examine the structure of the two simple gadgets in detail. Both gadgets contain an outermost condition that depends on whether the second argument is the special constant %, termed the *control token* (the first argument is always a continuation). Presence of the control token signifies that the value being presented is a non-gadget, such as the integer 1, and will appear as the third argument (to be bound by λV in the constant gadget and λB in the variable gadget). If the control token is not present, the second argument is itself the value being presented and will always be a gadget.

An alternate encoding, that would perhaps be clearer, would always use the second argument as a two-state control token, with the third argument as the value being presented. In a similar vein, on a tagged architecture the control token could be encoded as part of the tag, so that both the control information and the value presented would be obtained from the second argument.

Unification, then, is effected by application of the gadget representing one term to the gadget representing a second term, with a continuation always present as an additional argument:

(<term1> cont <term2>)

Constant gadgets

In an invocation like the one above, suppose the gadget in the function position encodes the constant 1.

(<1> cont <term2>)

Then the constant gadget effects a transfer of control from itself to the gadget in the argument position by applying that argument gadget to the continuation it has itself

received, the control token, and the constant value it, as a gadget, encodes. For example, in the *else* branch of <1>, the variable *N* is bound to the continuation received from the caller, the variable *A* is bound to the argument gadget, and the value encoded by the gadget is 1. The resulting call from the *else* branch following the application is

(<term2> cont % 1)

Note that the gadget <term2> will then necessarily take the *then* branch of the outermost condition.

The role of the *then* branch in the constant gadget is considerably simpler. It expects a non-gadget value to be presented and binds it to the variable *V*. It then tests for equality with the value it itself encodes, and executes the continuation or returns with failure depending on satisfaction of the equality.

Applying <1> to itself demonstrates the behavior of both branches:

(<1> cont <1>)

which, substituting for <1> in the function position, reduces by successive steps

```
((λN.λA.
  if (A==%)
    then (λV.if (V==1)
              then N
              else fail)
    else (A N % 1))
  cont
  <1>)
```

```
(if (<1>==%)
  then (λV.if (V==1)
                then cont
                else fail)
  else (<1> cont % 1))
```

(<1> cont % 1)

Substituting again for <1>, we have

```

((λN.λA.
  if (A==%)
    then (λV.if (V==1)
              then N
              else fail)
    else (A N % 1))
  cont
  %
  1)

```

```

((if (%==%)
  then (λV.if (V==1)
                then cont
                else fail)
  else (% cont % 1)))
  1)

```

```

((λV.if (V==1)
  then cont
  else fail)
  1)

```

```

(if (1==1)
  then cont
  else fail)

```

cont

which may be read as, "Execute the continuation *cont* as given," and is the expected result.

It should also be evident from a minor reworking of this example why (*<1> cont <2>*) is guaranteed to return *fail*.

Variable gadgets differ from constant gadgets by their inclusion of a *canonical representative*. For each logical variable *X* in the original program, there exists a distinct variable gadget *<X>* with its own canonical representative *X_c*. *X_c* can be thought of as capturing the state of the variable: it either has the value UNBOUND or is bound to another gadget that represents the value of the corresponding variable. We will have more to say about this subsequently.

Operationally, variable gadgets quite simple. If a gadget in the function position represents a variable, and no control token is presented as its argument, the argument presented is a gadget and one of the *else* branches is executed. The first of these traps the case in which the gadget is presented with itself (unification of a variable with itself), and as would be expected the gadget simply invokes its continuation. In the second *else* branch, if X_c , the canonical representative for the encoded variable X , is *UNBOUND*, then any free instances of X_c in the continuation are bound to the presented (gadget) value. Otherwise, in the final *else* clause, if X_c already has a gadget as its value, this gadget is applied to the presented value to test for unifiability.

The *then* branch of the variable gadget mirrors the final two *else* branches. Recall that since the control token is present, a non-gadget value, say the integer 1, is the presented value. If X_c is *UNBOUND*, then B is bound to 1, and a constant gadget is created with the value 1 embedded in it, i.e., in the $(\lambda N1.\lambda A. \dots)$ sub-expression. All occurrences of X_c in N are then bound to the newly created gadget, $\langle 1 \rangle$. If, on the other hand, X_c is already bound to a gadget, that gadget is applied to the (non-gadget) presented value. Note that the expression $(X_c N \%)$ will absorb the waiting third argument: as written it is equivalent to $(\lambda B.(X_c N \% B))$.

So, excepting the case of checking for unification with itself, the variable gadget does one of two things: if its canonical representative, X_c , is *UNBOUND*, it binds the gadget representation of the value presented to X_c in the continuation. If the canonical representative is bound, its value, a gadget, is applied to the gadget representation of the value presented to check for unifiability, and to effect further binding if necessary. The term X_c is always either *UNBOUND* or a gadget.

As a demonstration of the variable gadget, we present the unification problem that is symmetric with our initial example, i.e., the unification of 1 with X :

$(\langle 1 \rangle \text{ cont } \langle X \rangle)$

which, substituting for $\langle 1 \rangle$, gives

```
((\lambda N.\lambda A.
  if (A==%)
    then (\lambda V.if (V==1)
      then N
      else fail)
    else (A N \% 1))
  cont
  \langle X \rangle)
```



```

(if (<X>==%)
  then (λV.if (V==1)
             then cont
             else fail)
  else (<X> cont % 1))

```

```

(<X> cont % 1)

```

which, substituting for <X>, gives

```

((λN.λZ.
  if (Z==%)
    then if (Xc==UNBOUND)
      then (λB.(BIND Xc
                  (λN1.λA.
                    if (A==%)
                      then (λV.if (V==B)
                                   then N1
                                   else fail)
                      else (A N1 % B))
                  N))
      else (Xc N %)
    else if (Z==<X>)
      then N
    else if (Xc==UNBOUND)
      then (BIND Xc Z N)
    else (Xc N Z))
  cont
  %
  1)

```

```

((if (%==%)
  then if (Xc==UNBOUND)
    then (λB.(BIND Xc
      (λN1.λA.
        if (A==%)
          then (λV.if (V==B)
            then N1
            else fail)
          else (A N1 % B))
      cont))
    else (Xc cont %))
  else if (%==<X>)
    then cont
  else if (Xc==UNBOUND)
    then (BIND Xc % cont)
  else (Xc cont %))
1)

```

```

((if (Xc==UNBOUND)
  then (λB.(BIND Xc
    (λN1.λA.
      if (A==%)
        then (λV.if (V==B)
          then N1
          else fail)
        else (A N1 % B))
    cont))
  else (Xc cont %))
1)

```

which, under the assumption that *Xc* is *UNBOUND*, reduces to

```

((λB.(BIND Xc
  (λN1.λA.
    if (A==%)
      then (λV.if (V==B)
        then N1
        else fail)
      else (A N1 % B))
  cont))
1)

```

```

(BIND Xc
  (λN1.λA.
    if (A==%)
      then (λV.if (V==1)
        then N1
        else fail)
      else (A N1 % 1))
  cont)

```

in which the second argument is exactly <1>, so the expression is

```
(BIND Xc <1> cont)
```

and reduces to

```
[<1>/Xc]cont
```

as expected.

3.2.2. Structure gadgets

Gadgets encoding complex terms, or structures, are somewhat more complicated. We introduce structure gadgets in two stages, first defining gadgets for user-defined structures in terms of list-structure gadgets, and then showing how to define gadgets encoding list structures. Introduction of these additional gadgets also necessitates revision of the constant and variable gadgets introduced earlier, although the revisions are modest.

We remind the reader of a common scheme for encoding list constructors and selectors in the λ calculus:

```

(cons x y) = λS.Sxy
(first list) = (list (λX.λY.X))
(second list) = (list (λX.λY.Y))

```

In what follows, these functions play two roles: first, λ -expressions such as these are used as part of the gadget definitions, and *cons*, *first*, and *second* are used as shorthand names for the corresponding expressions; and second, when expressions of this form are present, they permit implementation via whatever list-structure support is provided in the target implementation.

Gadgets for user-defined structures

A user-defined structure can be implemented by the following gadget:

```
(λN.λA.  
  if (A==%)  
    then (λV.if (V==<functor>)  
            then (N <args>)  
            else fail)  
    else (A (λS.SN<args>) % <functor>)))
```

so that, for example, the gadget for a user structure *struct*(A,1) is written as

```
(λN.λA.  
  if (A==%)  
    then (λV.if (V==struct)  
            then (N <[A,1]>)  
            else fail)  
    else (A (λS.SN<[A,1]>) % struct))
```

where $\langle[A,1]\rangle$ signifies the gadget for the list structure $[A,1]$. Note that in this definition the form of continuations created and used differs from that employed in the previously introduced simple constant and variable gadgets. This matter will be addressed later.

List-structures gadgets

The primary role of list gadgets is to decompose the lists they encode, so that an expression for the unification of two lists:

$$(\langle[1,Y]\rangle N \langle[X,2]\rangle)$$

has the same effect as

$$(\langle 1 \rangle (\langle[Y]\rangle N \langle[2]\rangle) \langle X \rangle)$$

For the term $\langle[1,Y]\rangle$, for example, this gadget definition suffices:

```

(λN.λA.
  if (A==%)
    then (λV.if (V==cons)
      then (N <1> <[Y]>)
      else fail)
    else (A (λS.λT.(S (T N <[Y]>) <1>))) % cons))

```

Once again, the code constructing and making use of continuations differs from that provided for simple constant and variable gadgets, and for structure gadgets.

One approach to providing compatible definitions for all gadgets is to coerce all gadgets to binary form by inserting `<unique>` gadgets in slots that are otherwise unused. The new (and final) variable gadget becomes

```

(λN.λZ.
  if (Z==%)
    then if (Xc==UNBOUND)
      then (λB.(BIND Xc
        (λN1.λA.
          if (A==%)
            then (λV.if (V==B)
              then (N1 (N second first)
                (N first second))
              else fail)
            else (A (λS.λT.(S (T(N1)(N first second))
              (N second first)))
              %
              B))
          (N first first))))
    else (Xc N %)
  else if (Z==<X>)
    then N
  else if (Xc==UNBOUND)
    then ((λXc.N) Z)
  else (Xc N Z))

```

and the constant and structure gadgets are similarly revised:

```

(λN.λA.
  if (A==%)
    then (λV.if (V==1)
      then (N <unique> <unique>)
      else fail)
    else (A (λS.λT.(S (T N <unique>) <unique>))) % 1))

```

```

(λN.λA.
  if (A==%)
    then (λV.if (V==struct)
      then (N <unique> <[args]>)
      else fail)
    else (A (λS.λT.(S (T N <[args]>) <unique>))) % struct))

```

Note that the encoding of *<unique>* should be like that of the original constant gadget because substituting it for 1 in the definition above would make the definition recursive. This encoding is permissible because it only needs to be unifiable with itself and is never reconstructed by a variable gadget.

An alternative approach to defining gadgets

An alternative approach to defining constant and structure gadgets leads to more efficient implementations. It unfortunately is also more difficult to present. Instead of coercing all non-variable gadgets to binary constructors, it incorporates nullary (constant), unary (user structure), and binary constructor (list) gadgets. Constructor arity is encoded in the control token, so the resulting non-variable gadgets will successfully unify only with non-variable gadgets of the same arity. A modest degree of additional complexity arises in the code for the variable gadget, which must handle the different non-variable gadgets as distinct cases. The length of execution paths through the variable gadget remain nearly the same, however, and the performance benefits come from not having to unify *<unique>* gadgets that merely serve as place holders.

An example of structure unification

As an example of structure unification, we work through the unification of *struct(1,t(X))* and *struct(Y,t(2))*:

```

(<struct(1,t(X))> cont <struct(Y,t(2))>)

```

```

((λN.λA.
  if (A==%)
    then (λV.if (V==struct)
      then (N <unique> <[1,t(X)]>)
      else fail)
    else (A (λS.λT.(S (T N <[1,t(X)]>) <unique>)) % struct))
  cont
  <struct(Y,t(2))>)

```

```

(if (<struct(Y,t(2))>==%)
  then (λV.if (V==struct)
    then (cont <unique> <[1,t(X)]>)
    else fail)
  else (<struct(Y,t(2))> (λS.λT.(S (T cont <[1,t(X)]>) <unique>)) % struct))

```

```

(<struct(Y,t(2))> (λS.λT.(S (T cont <[1,t(X)]>) <unique>)) % struct)

```

```

((λN.λA.
  if (A==%)
    then (λV.if (V==struct)
      then (N <unique> <[Y,t(2)]>)
      else fail)
    else (A (λS.λT.(S (T N <[Y,t(2)]>) <unique>)) % struct))
  (λS.λT.(S (T cont <[1,t(X)]>) <unique>))
  %
  struct)

```

```

((if (%==%)
  then (λV.if (V==struct)
    then ((λS.λT.(S (T cont <[1,t(X)]>) <unique>))
      <unique>
      <[Y,t(2)]>)
    else fail)
  else (% (λS.λT.(S (T (λS.λT.(S (T cont <[1,t(X)]>) <unique>))
    <[Y,t(2)]>)
    <unique>))
    %
    struct))
  struct)

```

```

((λV.if (V==struct)
  then ((λS.λT.(S (T cont <[1,t(X)]>) <unique>)) <unique> <[Y,t(2)]>)
  else fail)
 struct)

```

```

(if (struct==struct)
  then ((λS.λT.(S (T cont <[1,t(X)]>) <unique>)) <unique> <[Y,t(2)]>)
  else fail)

```

```

((λS.λT.(S (T cont <[1,t(X)]>) <unique>)) <unique> <[Y,t(2)]>)

```

```

(<unique> (<[Y,t(2)]> cont <[1,t(X)]>) <unique>)

```

which reduces, according to the definition of <unique>, to

```

(<[Y,t(2)]> cont <[1,t(X)]>)

```

```

((λN.λA.
  if (A==%)
    then (λV.if (V==cons)
      then (N <Y> <[t(2)]>)
      else fail)
    else (A (λS.λT.(S (T N <[t(2)]>) <Y>)) % cons))
 cont
 <[1,t(X)]>)

```

```

(if (<[1,t(X)]>==%)
  then (λV.if (V==cons)
    then (cont <Y> <[t(2)]>)
    else fail)
  else (<[1,t(X)]> (λS.λT.(S (T cont <[t(2)]>) <Y>)) % cons))

```

```

(<[1,t(X)]> (λS.λT.(S (T cont <[t(2)]>) <Y>)) % cons)

```



```

((λN.λA.
  if (A==%)
    then (λV.if (V==cons)
      then (N <1> <[t(X)]>)
      else fail)
    else (A (λS.λT.(S (T N <[t(X)]>) <1>)) % cons))
(λS.λT.(S (T cont <[t(2)]>) <Y>))
%
cons)

```

```

((if (%==%)
  then (λV.if (V==cons)
    then ((λS.λT.(S (T cont <[t(2)]>) <Y>)) <1> <[t(X)]>)
    else fail)
  else (% (λS.λT.(S (T (λS.λT.(S (T cont <[t(2)]>) <Y>)) <[t(X)]>) <1>)) % cons))
cons)

```

```

((λV.if (V==cons)
  then ((λS.λT.(S (T cont <[t(2)]>) <Y>)) <1> <[t(X)]>)
  else fail)
cons)

```

```

(if (cons==cons)
  then ((λS.λT.(S (T cont <[t(2)]>) <Y>)) <1> <[t(X)]>)
  else fail)

```

```

((λS.λT.(S (T cont <[t(2)]>) <Y>)) <1> <[t(X)]>)

```

```

(<1> (<[t(X)]> cont <[t(2)]>) <Y>)

```

which we have shown for earlier versions of the constant and variable gadgets, and under the assumption that Y_c is *UNBOUND*, to reduce to

```

[<1>/ $Y_c$ ]
(<[t(X)]> cont <[t(2)]>)

```

```

[<1>/Yc]
((λN.λA.
  if (A==%)
    then (λV.if (V==t)
      then (N <unique> <[X]>)
      else fail)
    else (A (λS.λT.(S (T N <[X]>) <unique>)) % t))
cont
<[t(2)]>)

```

```

[<1>/Yc]
(if (<[t(2)]>==%)
  then (λV.if (V==t)
    then (cont <unique> <[X]>)
    else fail)
  else (<[t(2)]> (λS.λT.(S (T cont <[X]>) <unique>)) % t))

```

```

[<1>/Yc]
(<[t(2)]> (λS.λT.(S (T cont <[X]>) <unique>)) % t)

```

```

[<1>/Yc]
((λN.λA.
  if (A==%)
    then (λV.if (V==t)
      then (N <unique> <[2]>)
      else fail)
    else (A (λS.λT.(S (T N <[2]>) <unique>)) % t))
(λS.λT.(S (T cont <[X]>) <unique>))
%
t)

```

```

[<1>/Yc]
((if (%==%)
  then (λV.if (V==t)
    then ((λS.λT.(S (T cont <[X]>) <unique>)) <unique> <[2]>)
    else fail)
  else (% (λS.λT.(S (T (λS.λT.(S (T cont <[X]>) <unique>)) <[2]>) <unique>))
    %
    t))
  t)

```

```

[<1>/Yc]
((λV.if (V==t)
  then ((λS.λT.(S (T cont <[X]>) <unique>)) <unique> <[2]>)
  else fail)
  t)

```

```

[<1>/Yc]
(if (t==t)
  then ((λS.λT.(S (T cont <[X]>) <unique>)) <unique> <[2]>)
  else fail)

```

```

[<1>/Yc]
((λS.λT.(S (T cont <[X]>) <unique>))
  <unique>
  <[2]>)

```

```

[<1>/Yc]
(<unique> (<[2]> cont <[X]>) <unique>)

```

which we know reduces to

```

[<1>/Yc]
(<[2]> cont <[X]>)

```

and we can show by an analogous sequence that this in turn reduces to

```

[<2>/Xc,<1>/Yc]
cont

```

which is the expected result.

3.2.3. Non-functional operations in gadget definitions

The *BIND* operation cannot be directly effected using β reduction in the λ calculus. For example, the expression

$$((\lambda N.((\lambda Xc.N) Z)) \text{ cont})$$

fails to achieve the desired result (of substituting Z for Xc in *cont*) because it would depend on "capture" of free occurrences of Xc in *cont* as part of the outermost reduction, and such a mechanism suffices to introduce inconsistency into the λ calculus [2].

While continuation passing plays an important role in implementing the coroutining procedure, of greater import is the fact, reflected here, that in a *BIND* expression the continuation captures exactly the intended binding scope of the corresponding logical variable. So *BIND* and continuation passing together play a central role in defining variable gadgets, and yet the *BIND* operation clearly lies outside the λ calculus.

An issue we have not addressed is the creation and "initialization" of what we have termed the canonical representatives of a logical variable, e.g., λ variable Xc for logical variable X . This is a significant issue since logic-programming procedures, unlike functional procedures, allocate variables on each invocation, and the scope and lifetime of a logical variable may exceed that of the procedure instance that creates it. In section 3 we will demonstrate one solution, adopted in our implementation, that uses a non-deterministic *ALLOCATE* combinator.

We also have not explicitly addressed the question of how multiple solutions are captured in the target language. Spawning of multiple solutions is achievable using either the continuation or proof-stream methods, as reported by Carlsson [3] and Nilsson [20], and this issue is largely independent of the gadget technique. Management of *BIND*able terms across multiple solutions is more problematic. There are two well-known solutions: trailing and unbinding of *BIND*able terms (logical variables) in the case that solutions are generated depth-first, and copying of *BIND*able terms for breadth-first search strategies. Either, in conjunction with suitable methods for implementing *BIND*, is fully compatible with the gadget technique; both also fall clearly outside the bounds of a purely functional definition.

3.2.4. Correctness, parallelisability, and evaluation policies

The gadget definitions we have given are correct with respect to an operational definition of unification under a policy of normal order evaluation. This can be shown by examining all pairings of the individual gadget types as part of an inductive argument.

Correctness is not necessarily preserved under other evaluation policies. Considering what would happen if two gadgets encoding the same logical variable were to pass the " $(X_c == UNBOUND)$ " test before either performed the subsequent *BIND* operation. That is, one might execute " $(BIND\ X_c <1> N)$ " while the other executed " $(BIND\ X_c <2> N)$ " and there would be no "check" for unifiability between the two bindings. The loss of the Church-Rosser property is of particular concern given our goal of parallel implementation.

The solution is to eliminate execution sequences such as the one above by requiring exclusive access within both test-*BIND* sequences in the variable gadget. The implementation cost will depend on the underlying architecture, but on uniprocessor or shared-memory machines, one can expect this to be implemented using a read-modify-write instruction. As an architectural bottleneck this is expected to be benign, since exclusion occurs only on simultaneous attempts to be the *first* process to access a particular instance of a canonical representative, X_c .

3.2.5. Additional perspectives on gadgets

It is worth mentioning both the object-oriented view of gadgets and the relationship of gadgets to the special-purpose unification instructions provided by the Warren Abstract Machine [24] and POPLOG Virtual Machine [16, 17].

Gadgets are encodings of terms, and from an object-oriented point of view they are objects that capture the states and methods relevant to unification that are associated with the term they encode [9]. These include methods for effecting unification, querying/reporting the type and value of the associated term, selecting the appropriate continuation depending on equality test results, and storing the state (unbound/bound and value) in the case that the associated term is a logical variable.

Gadgets also bear a resemblance to the special-purpose unification instructions defined for the PROLOG and POPLOG virtual machines, defined respectively by Warren [24] and by Mellish and Hardy [16, 17]. In both of these schemes, unification is encoded in-line as far as the definition of formal arguments permits, using the special-purpose instructions. The resulting code always comprises the header of a compiled clause definition.

The difference between this approach and ours demonstrates a fundamental property of gadgets. Using the gadget technique, similar special-purpose code, in the form of higher-order functions, is associated with each term in a program rather than with the head of each clause definition. As a consequence, clause header code generated using the gadget technique consists only of applying formal arguments to actual arguments, with the code for the clause body appearing as the continuation (examples are shown in section 3). Since formal and actual arguments are both gadgets, this operation applies the effects of unifying the corresponding terms to the body of the clause definition. Gadget code is therefore used to implement all unification as efficient special-purpose code rather than just that that corresponds to unification with a statically-declared formal argument. A further benefit, as we shall see in section 4, is

that coding the gadgets in an almost-functional language allows the resulting code to be manipulated using powerful program transformation and compilation techniques developed for functional languages. A body of special-purpose methods need not be defined, and explicit unification no longer constitutes a barrier to moving code "across" unification.

Finally, we note that the gadget technique is entirely compatible with several existing implementation methods:

- Gadget code is susceptible to lifetime analysis, so that efficient allocation from the stack is possible just as in the Warren Abstract Machine.
- While our combinator-based prototype implements *BIND* as an overwriting operation, implementation in a closure-based system would be handled in the obvious way. Consequently, compilation technology developed for Scheme and T [14] is applicable to an implementation making use of gadgets.
- The gadget technique places no requirements upon the methods used to manage logical-variables/canonical representatives in the context of multiple-solution languages. Consequently, gadgets can be used to implement depth-first or breadth-first languages, and can make use of existing or as-yet undeveloped solutions to this problem.

3.3. Implementation via combinator reduction

Our combinator-based implementation consists of an SKI interpreter extended to incorporate combinators that implement the gadgets defined in the previous section. Programs are run through a "compiler" that translates Juniper's PROLOG-like clause definitions to SASL-like functional programs, hereafter termed *gadget code*, in which gadget combinators appear explicitly. This code can then be loaded and run by the interpreter. In what follows, we present programs as gadget code, rather than the SKI expressions manipulated by the interpreter internally, because the latter are opaque and the relationship between the two is well understood.

In the remainder of this section, we discuss the issue of combinator granularity for the implementation of gadgets, and then move on to definitions of the combinators. As a broader exposition of the implementation, we subsequently examine gadget code from the standpoint of the four implementation sub-problems identified in the introduction.

Note that while the following sections are based on the gadget definitions discussed earlier, which treat all non-variable terms as binary constructors, our implementation actually uses the alternate, more efficient definitions of gadgets outlined in the section *An alternative approach to defining gadgets* above.

3.3.1. Implementing gadgets via combinators

The principal issue governing the choice of a combinator set to support the implementation of gadgets is the granularity at which gadgets are to be implemented. One obvious approach would be to implement *BIND* and *ALLOCATE* (to be discussed below) as primitive operators and treat the body of the gadget definitions as functional code.

There are two benefits of choosing an alternative approach, however, in which one large-grained combinator is defined for each of the gadgets introduced earlier. First, performance is enhanced by pushing the gadgets down one level of interpretation, so that execution of a gadget is performed in a single interpreted operation; and second, the larger granularity allows the interpreter to exploit the lifetime properties of cells allocated during gadget execution, so that a single level of unification requires at most two, and more typically zero or one, allocations. We expect the latter property to carry over to compiled implementations, so large-grained combinators compiled by special-purpose routines may be suitable in implementations that do allocation predominantly from the heap, while compilation as general-purpose functional code, in conjunction with *BIND* and *ALLOCATE* primitives, may be suitable for implementations that do allocation from the stack.

3.3.2. Gadget definitions

The six combinators used to implement gadgets are defined as follows:

```
(ALLOCATE n proc)
  => (proc alloc_1 ... alloc_n)
```

where *alloc_i* is a unique constant term *UNBOUND* allocated as the canonical representative *X_c* of a logical variable *X*. *ALLOCATE* is non-functional because it returns a different expression (and set of *alloc_i*'s) each time it is called. The role of *ALLOCATE* is discussed in the section below, Implementation of logical variables.

```
(VAR_GADGET canonical_rep cont <gadget>)
  => [<gadget>/canonical_rep|cont
(VAR_GADGET <gadget1> cont <gadget2>)
  => (<gadget1> cont <gadget2>)
(VAR_GADGET var [hd.tl.cont] % constr)
  => [(STRUCT_GADGET constr hd tl)/var|cont
(VAR_GADGET <gadget> [hd.tl.cont] % value)
  => (<gadget> [hd.tl.cont] % value)
```

Note that *VAR_GADGET* subsumes the *BIND* operation.

```
(STRUCT_GADGET constr hd tl cont <gadget>)
```

```

=> (<gadget> [hd.tl.cont] % constr)
(STRUCT_GADGET constr1 hd1 tl1 [hd2.tl2.cont] % constr1)
=> (hd1 (tl1 cont tl2) hd2)
(STRUCT_GADGET constr1 hd1 tl1 [hd2.tl2.cont] % constr2)
=> NIL

```

```

(COPY_UNBOUND (f cont arg1 ... argn))
=> (f cont arg1' ... argn')

```

argi' is a copy of *argi* with a new *UNBOUND* constant substituted for each existing *UNBOUND* constant.

```

(UNIQUE cont UNIQUE)
=> cont

```

UNIQUE implements the *<unique>* slot-filler gadget.

Using these combinators, the different gadgets are defined as

```

<1> = (STRUCT_GADGET 1 UNIQUE UNIQUE)
<X> = (VAR_GADGET Xc)
<struct(1)> = (STRUCT_GADGET struct
               UNIQUE
               (STRUCT_GADGET 1 UNIQUE UNIQUE))
<[X|Y]> = (STRUCT_GADGET cons (VAR_GADGET Xc) (VAR_GADGET Yc))

```

3.3.3. Gadget code

The widely-cited (and incorrect) logic-programming procedure for *append()* is defined in Prolog by

```

app([],X,X).
app([H|T],X,[H|R]) :- app(T,X,R).

```

(*app()* is incorrect because it includes *cons()* in its extension, i.e., *cons(A,B,ConsStructure) :- app([A],B,ConsStructure)*). The corresponding gadget code

```

app3 cont arg1 arg2 arg3
=
  (APPEND (COPY_UNBOUND (app3_1 cont arg1 arg2 arg3))
          (app3_2 cont arg1 arg2 arg3))

```



```

app3_1 = (ALLOCATE 1 _app3_1)
_app3_1 alloc_1 cont arg1 arg2 arg3
=
  ((STRUCT_GADGET NIL UNIQUE UNIQUE)
   ((VAR_GADGET alloc_1)
    ((VAR_GADGET alloc_1)
     cont
     arg3)
    arg2)
   arg1)

app3_2 = (ALLOCATE 4 _app3_2)
_app3_2 alloc_1 alloc_2 alloc_3 alloc_4 cont arg1 arg2 arg3
=
  ((STRUCT_GADGET
   cons
   (VAR_GADGET alloc_1)
   (VAR_GADGET alloc_2))
   ((VAR_GADGET alloc_3)
    ((STRUCT_GADGET
     cons
     (VAR_GADGET alloc_1)
     (VAR_GADGET alloc_4))
     (app3
      cont
      (VAR_GADGET alloc_2)
      (VAR_GADGET alloc_3)
      (VAR_GADGET alloc_4))
      arg3)
     arg2)
    arg1)

```

addresses all four of the implementation issues identified in the introduction. These will be discussed individually in the following sections.

Translating logic-programming control constructs to gadget code

The entry point to this code is the function *app3()*. Note the presence of the continuation argument, which must be passed in turn to children spawned by the function. The body of the definition collects multiple solutions, *APPENDING* together all solutions returned by the functions implementing the two original clauses of *app()*. Under our lazy evaluation policy, this code produces a left-to-right depth-first search.

We are currently modifying our implementation, extending the proposal put forth by Hughes [11], to do concurrent graph reduction. This will allow substitution of an associative, commutative bag constructor for APPEND in the body of this top-level function and provide breadth-first evaluation, with bag elements returned in the order they are computed (this semantics for bag generators is discussed by Reddy [21]).

The other major control construct in Prolog is the infamous *cut()*. Since the database is unordered in Juniper, *cut()* is not meaningful, and we have instead implemented an *if_then_else()* operator. *if_then_else()* is similar to *cut()*, except that its scope is restricted to the body of the clause in which it appears, and its semantics is such that it may generate solutions in the *then* branch for each solution of its condition, rather than for just the first, as *cut()* would.

if_then_else() is defined in terms of *call()* and *not()* (which we rename *fail()*), so implementation of *call()* and *fail()* suffices to provide all necessary control constructs. Within the context of the full Juniper language, *call()* and *fail()* are easily implemented, although the details are beyond the scope of this paper.

Mapping unification to gadget code

As discussed at length in section 2, unification does not appear explicitly in gadget code. Every logic-programming clause is compiled to gadget code of the form

```
(<formal>
  cont
  arg)
```

where *<formal>* is the gadget encoding a formal argument of the clause and *arg* will be bound to an actual argument of the clause (also a gadget). For example, in *_app3_1()*, unification of the first formal with the first actual argument is effected by the outermost application of

```
((STRUCT_GADGET NIL UNIQUE UNIQUE)
  ((VAR_GADGET alloc_1)
    ((VAR_GADGET alloc_1)
      cont
      arg3)
    arg2)
  arg1)
```

with the sub-expression

```

((VAR_GADGET alloc_1)
  ((VAR_GADGET alloc_1)
    cont
    arg3)
  arg2)

```

as its continuation.

Mapping logic-programming variables to gadget code

The mapping of logical variables to gadget code has several elements. We have already discussed canonical representatives, which implement the unbound or bound state and value of the corresponding logical variables. We have also mentioned that *BIND*, which appears as a primitive in the gadget definitions, is implemented by overwriting the value of a canonical representative. This overwriting, in conjunction with continuation passing, serves to implement variable binding with the correct scope.

Two issues remain: the question of how canonical representatives are allocated, and the question of how overwriting of a particular canonical representative in one part of the code is communicated to its other occurrences in a program.

In a graph-reduction implementation, there is a natural way to address both questions that appears to have been discovered at least three times, by ourselves, Finn [6], and Bage and Lindstrom [1].

The latter problem is addressed by taking advantage of the term-sharing property provided by graph reduction implementations [23]. That is, for a function body *body* containing multiple references to a variable gadget (*VAR_GADGET Xc*), Turner's bracket abstraction algorithm computes an expression *body'* such that

$$\text{body} = (\text{body}' \text{ Xc})$$

In a graph-reduction implementation, reduction of the expression on the right-hand side causes *Xc*, the pointer to a single location in memory allocated for the canonical representative, to be "distributed" to all places in the body where it is referenced.

In the definition of *_app3_1()*, this is precisely the role that the argument *alloc_1* plays: it stands in for the canonical representative required in the body of the definition. The function *_app3_1()* is the invocation-independent gadget code for the first clause of *app()*, while the application of *_app3_1()* to a unique canonical representative the actual argument for *alloc_1*, produces invocation-specific gadget code for the clause.

We return to the first problem, that of allocating individual canonical representatives. Like Prolog, and in contrast with Id Nouveau [19], logical variables in Juniper are allocated implicitly as part of the call to a logic clause.

A solution immediately presents itself. Each invocation of `_app3_1()` needs to be supplied with a unique canonical representative, and this is accomplished by wrapping an `ALLOCATE` expression around every call to `_app3_1`, as shown in the definition of `app3_1`. The `ALLOCATE` combinator takes two arguments, the first being the number of canonical representatives and the second a function. At each invocation it supplies the required number of unique (freshly allocated) canonical representatives as initial arguments to the function argument. The resulting expression then awaits exactly the continuation and actual arguments that are to be supplied by the calling procedure. `ALLOCATE`'s non-deterministic behavior proved to be quite simple to implement in our interpreter.

One restriction that this method of implementing logical variables places on the interpreter should be noted. Normally, sharing improves performance by eliminating recomputation of shared expressions. Should a shared term be copied, however, recomputation affects neither the termination properties nor the denotation of the program. In our implementation, sharing must be preserved, so that the overwriting of a canonical representative is successfully communicated to all the expressions that reference it. Copying after overwriting remains permissible.

Support for logical variables with multiple solutions

There are two conventional methods for managing logical variables, or in this case canonical representatives for variable gadgets, when multiple children need to bind an unbound variable inherited from their parent. The first, of which there are several variants, is to execute one child at a time, remembering which bindings are made. When execution is completed and control is transferred to the next child, all bindings from the first are undone. This provides the depth-first search strategy required by Prolog. The second method consists of copying all unbound variables on spawning, so that each child will in fact be binding a different copy of the variable inherited from its parent. This method can be used to provide breadth-first search. Both methods are compatible with the gadget technique.

The current implementation uses copying, implemented by the `COPY_UNBOUND` operator, to create a new instance of each `UNBOUND` canonical representative occurring in arguments to the child. In the definition of `app3()` above, copying is done for the left child, while the right child inherits the originals from the parent.

3.4. Gadgets and optimising transformations

We have suggested that the program transformation and implementation techniques developed for functional languages will find applicability for gadget code as well. The validity of this claim is a subject for further study. As a preface to that work, we discuss here four optimizing transformations that are attainable, and demonstrate a simple but powerful technique for local optimization of gadget code that subsumes three of the four.

There are a number of general-purpose optimizing transformations for functional programs that are desirable and attainable. Examples are code motion out of a loop and fold/unfold transformations, which are demonstrated in [23] in the context of combinator-based implementations, and occur in more sophisticated form in Hughes' work on supercombinators [12] and as an application of the abstract interpretation technique. We believe that these techniques will be directly applicable also to gadget code, but do not pursue them further here.

3.4.1. Permissiveness of gadget code

One observation that is central to investigating program transformations, and interesting as a property of gadget code in its own right, is that under restrictions discussed below, the *BIND* operation is *permissive* with respect to transformations that are designation-preserving when applied to purely functional code.

Assignment is a confounding factor in the characterisation of many programming languages, but the *BIND* operation in gadget code is much more limited even than assignment in a single-assignment language would be (on the view that functional languages are zero-assignment languages). This is because in gadget code only canonical representatives may be assigned to, and canonical representatives appear only within variable gadgets. A variable gadget has two behaviors: on the first occasion that it is invoked, it assigns the value of its argument to its canonical representative. On subsequent invocations, it tests the value of its argument against the value assigned to its canonical representative. If test-*BIND* sequences within variable gadgets are uninterruptable, as discussed in the subsection *Correctness*, *parallelisability*, and *evaluation policies* above, it makes no difference in which order two instances of a variable gadget are invoked. This result is not surprising, given the properties of logical variables and unification that this code implements.

The importance of this property to program transformations is immediate. Transformations do not change the behavior of the program by interchanging the order of invocation of variable gadgets, modulo issues of termination.

Of course, movement of a *BIND* operation across a *COPY_UNBOUND* operation is unsafe. Consequently, we will define local transformations as those that occur within the code bounded by *COPY_UNBOUND* operations, and concern ourselves only with local optimizing transformations.

3.4.2. Logic-programming-specific optimising transformations

Of particular interest are transformations that address the role of gadget code as implementing logic programs. To date we have identified four kinds of transformations that provide local optimizations: partial unification, conversion of depth-first to breadth-first unification, optimisation for mode declarations, and variable elimination. Three of these are subsumed by a single technique that we will summarize in the final sub-section, A uniform technique for local optimisation of gadget code. These

efforts bear some resemblance to the work reported by Kahn and Carlsson [13].

For consistency in our discussion of code optimization, we continue our practice of using gadget code. Note, however, that many of the optimizations described below entail expansion of gadgets at a finer grain than is provided by the large-grain gadget combinators.

Although we will not discuss them here, it is worth noting that techniques such as abstract interpretation have been applied in an *ad hoc* fashion to logic programs [4, 17, 18], and one of our goals is a more accessible and effective application of the same techniques to gadget code.

Partial unification

Partial unification can be achieved as a side effect of partial evaluation applied to the gadget code. Application of a constructor gadget appearing as a formal argument to a presumptive actual argument

```
(STRUCT_GADGET constr hd tl
  cont
  argn)
```

may be evaluated at compile time to yield

```
(argn [hd.tl.cont] % constr)
```

Similarly, for a variable gadget appearing as a formal argument, the expression

```
(VAR_GADGET Xc
  cont
  argn)
```

evaluates to

```
[argn/Xc]cont
```

if X_c is unbound, which can be inferred since it is appearing as a formal argument. If X_c is bound, the evaluation continues with

```
(<Y> cont argn)
```

where $\langle Y \rangle$ is the value of X_c .

Nested gadget applications, of course, can be expanded as well.

Breadth-first unification

The Warren Abstract Machine provides depth-first unification, whether implemented as special-purpose code or using the general-purpose unification procedure. One consequence is that the ordering of arguments may markedly affect performance, since unification failure occurs frequently in the execution of Prolog programs. By contrast, breadth-first unification provides near-optimal early failure irrespective of the order of arguments. Breadth-first unification of formal arguments is easily obtained in gadget code through a simple reordering of the function body.

Optimisation for mode declarations

Mode declarations generally enable optimisations in the implementation of Prolog code. The declaration of modes permits particular functional dependencies to be inferred, and these dependencies can be captured in the gadget code for a clause. Given, for example, the unmoded code for the second clause of *app()* above,

```
_app3_2 alloc_1 alloc_2 alloc_3 alloc_4 cont arg1 arg2 arg3
=
  ((STRUCT_GADGET
    cons
    (VAR_GADGET alloc_1)
    (VAR_GADGET alloc_2))
  ((VAR_GADGET alloc_3)
    ((STRUCT_GADGET
      cons
      (VAR_GADGET alloc_1)
      (VAR_GADGET alloc_4))
    (app3
      cont
      (VAR_GADGET alloc_2)
      (VAR_GADGET alloc_3)
      (VAR_GADGET alloc_4))
    arg3)
  arg2)
  arg1)
```

if the first and second arguments are declared as ground and the third argument is a variable, then partial unification, together with the knowledge that all *alloc_i*'s are within scope directly produces the optimized definition

```
_app3_2 alloc_1 alloc_2 alloc_3 alloc_4 cont arg1 arg2 arg3
```

```

if ((constr_g arg1) == cons)
  then
    (arg3
      (app3
        cont
        (tl_g arg1)
        arg2
        (VAR_GADGET alloc_4))
      (STRUCT_GADGET
        cons
        (hd_g arg1)
        (VAR_GADGET alloc_4)))
  else NIL

```

where *constr_g*, *hd_g* and *tl_g* are the gadget analogues of conventional selectors. The "trick" that allows these functional dependencies to be captured is substitution of the expression

```
(STRUCT_GADGET (constr_g arg1) (hd_g arg1) (tl_g arg1))
```

for *arg1* prior to the partial evaluation.

Variable elimination

Variable elimination is important not only as a program optimization, but as a step toward the goal of investigating the gap between logic-programming and functional languages.

Intuitively, the first clause for Prolog *app()*

```
app([],X,X).
```

need not allocate the logical variable *X* — it would suffice to simply unify the second and third actual arguments. And indeed, partial evaluation of the initial gadget code


```

_app3_1 alloc_1 cont arg1 arg2 arg3
=
((STRUCT_GADGET NIL UNIQUE UNIQUE)
  ((VAR_GADGET alloc_1)
    ((VAR_GADGET alloc_1)
      cont
      arg3)
    arg2)
  arg1)

```

yields code that appears to have the desired form

```

_app3_1 alloc_1 cont arg1 arg2 arg3
=
(arg1
  (λS.λT.
    (S
      (T
        (arg2
          cont
          arg3)
        UNIQUE)
      UNIQUE))
  %
  NIL)

```

But this is not optimal because the canonical representative, *alloc_1* is allocated for, but not used in, *_app3_1*. Consequently, it is necessary to examine the code and determine which canonical representatives retain their *UNBOUND* values following partial evaluation, and revise the definition accordingly. We see, for example, that after eliminating unnecessary canonical representatives, the previously-optimized definition of the second clause becomes

```

_app3_2 alloc_4 cont arg1 arg2 arg3
=
  if ((functor arg1) == cons)
    then
      (arg3
        (app3
          cont
          (tl_g arg1)
          arg2
          (VAR_GADGET alloc_4))
        (STRUCT_GADGET
          cons
          (hd_g arg1)
          (VAR_GADGET alloc_4)))
    else NIL

```

which calls for allocation of one canonical representative rather than the original four.

A uniform technique for local optimisation of gadget code

What we have demonstrated in three previous sub-sections is a uniform technique for performing local program optimisations. Recall that we have been describing function definitions in terms of gadget code, but the actual definitions (in our implementation) are stored and executed as SKI expressions. The SKI expression defining a function is produced by Turner's bracket-abstraction algorithm, which, using the original function body, abstracts out formal arguments in the order in which they appear in the definition. The result is an expression, which, when applied to actual arguments, "distributes" actual arguments to the positions formerly occupied by the formal arguments.

The procedure implementing our optimisation technique has two stages. First, the function definition, obtained as an SKI expression, is applied to the designated number of canonical representatives followed by suitable symbolic arguments standing in for actual arguments. The whole expression is partially evaluated as far as possible without expanding function references. At this point, reordering of the resulting expression, to effect breadth-first unification or certain cases of variable-elimination, may be applied.

One side effect of the partial evaluation, as indicated above, is that some canonical representatives may be overwritten. When the partial evaluation is complete, the expression that remains is examined to identify any remaining *UNBOUND* canonical representatives. Those that remain are abstracted out in the order they originally appeared. The calling (*ALLOCATE...*) expression is then patched to reflect the revised number of canonical representatives. Finally, the remaining symbolic arguments are abstracted out in the order in which they appeared in the original definition. The

result is an optimized expression defining the original function, modulo deletion of some canonical-representative arguments. For the example of `_app3_2()` discussed above, the result is a combinator expression corresponding to the final optimized gadget-code definition listed just above.

As mentioned earlier, when mode declarations are provided, the technique is modified by using symbolic arguments that reflect the functional dependencies inferred from the declaration.

Implications of variable elimination

The ability to eliminate variable has interesting implications with regard to the differences between logic-programming and functional languages. We have shown that in certain cases it is possible to infer the value a logical variable will be assigned and eliminate it. We believe that the search for those cases are far from exhausted. If all logical variables could be eliminated, then the *ALLOCATE*, *BIND*, and *COPY_VARS* operations would be unnecessary, and we would have a purely functional program.

Similarly, if it can be statically inferred that a variable will never be bound, it may simply be treated (in the λ calculus) as a free variable. This provides a solution only if the number of free variables can be determined at compilation time.

Finally, we note that the number of free variables in a solution is less than may be commonly assumed, since there are identifiable functional dependencies among the variables. For example, given the goal

```
app([1|A],[],Result)
```

the interpreter returns a set of solutions

```
{ Result = [1], Result = [1,X001], Result = [1,X001,X002], ... }
```

but these can all be expressed in terms of the initial (free) variable *A* as

```
{ Result = [1], Result = [1,(hd A)], Result = [1,(hd A),(hd (tl A))], ... }
```

where *hd* and *tl* are the usual selector functions.

This leads us to the conjecture that if it can be statically determined that for each variable in a program, either the variable will necessarily be bound or the variable will necessarily remain unbound throughout the program and there are a fixed number of such variables, then the program can be directly translated to a purely functional language.

3.5. Conclusion

We have demonstrated a technique that permits translation of logic programs to an almost-functional form. We expect this to allow direct translation of logic programs to SCHEME or T, and to functional programming languages whose implementations have been modestly extended to support gadgets. Both the optimizing transformations outlined herein and the technology available for target implementations can therefore be brought to bear on the implementation of logic programs.

The translation of logic programs to functional programs is interesting in its own right, and we feel that the gadget technique provides a significant step toward that goal.

Our results suggest two principal directions for further study. First, we intend to further investigate techniques adopted from the study of functional languages. The goals are both to obtain efficient implementations and to push on with investigation of the language translation problem. Second, our work with gadgets has provided both a route to implementation and a better understanding of the relationship between functional and logic-programming languages. Both of these have contributed to our ongoing efforts to define additional members of the Juniper functional + logic-programming language family.

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Rome Air Development Center (RADC) of the Air Force Systems Command under contract F30602-86-C-0093.

The author wishes to acknowledge the many and varied contributions of Bill Hopkins, Lynette Hirschman, Don McKay, and Mickey Galatola to this work and to the MASC program at Unisys Corp. since its inception. In addition, Bob Smith, Clarke Arnold, and Massoud Farhang have provided able assistance throughout.

References

1. Bage, G. and Lindstrom, G., Combinator Evaluation of Functional Programs with Logical Variables, Tech. Rep. UUCS-87-027, Department of Computer Science, University of Utah, October 1987.
2. Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*, rev. ed., North-Holland, Amsterdam, 1984.
3. Carlsson, M., On Implementing Prolog in Functional Programming, *New Generation Computing* 2, (1984), 347-359, Ohmsha, Ltd. and Springer-Verlag.
4. Debray, S. K. and Warren, D. S., Automatic Mode Inference for Prolog Programs, in *Proceedings of 1986 IEEE Symposium on Logic Programming*, Salt Lake City, September 1986, 78-88.
5. Felleisen, M., Transliterating Prolog into Scheme, Tech. Rep. 182, Indiana University Computer Science Department, Oct 1985.
6. Finn, S., The Simplex Programming Language, Tech. Rep., Department of Computing Science, University of Stirling, March 1985.
7. Goldberg, B. and Hudak, P., Serial Combinators: Optimal Grains for Parallelism, in *Functional Programming Languages and Computer Architecture*, J. Jouannaud (ed.), Springer-Verlag, 1985, 382-399.
8. Greene, K. J., *User's Guide to the LNF-Plus System*, Syracuse University/Rome Air Development Center, January, 1987.
9. Gusman, J., *personal communication*, Yale University, 1987.
10. Haynes, C. T., Logic continuations, in *Proceedings of the Third Int'l. Conf. on Logic Programming*, July, 1985, London, England, vol. 225, Springer-Verlag, Berlin, 1985, 671-685.
11. Hughes, R. J. M., *A Simple Implementation of Concurrent Graph Reduction*, Graph Reduction Workshop, Santa Fe, New Mexico, Santa Fe, New Mexico, Sept. 30 - Oct. 1, 1986..
12. Hughes, R. J. M., Super-Combinators: A New Implementation Method for Applicative Languages, in *Proc. ACM Symposium on LISP and Functional Programming*, ACM, 1982, 1-10.
13. Kahn, K. M. and Carlsson, M., The compilation of Prolog programs without the use of a Prolog compiler, Technical report 27, University of Uppsala. Department of Computer Science. Uppsala Programming Methodology and Artificial Intelligence Laboratory, 1984.
14. Krans, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J. and Adams, N., ORBIT: An Optimising Compiler for Scheme, in *Proceedings SIGPLAN Notices '86 Symposium on Compiler Construction*, 1986.
15. Lindstrom, G., Functional programming and the logical variable, *Proc. Twelfth ACM Sympos. on Principles of Programming Languages*, New Orleans, 1985, 266-280.

16. Mellish, C. and Hardy, S., Integrating Prolog in the POPLOG Environment, in *Implementations of Prolog*, J. A. Campbell (ed.), Halsted Press/John Wiley, New York, 1984, 147-162.
17. Mellish, C. S., Some Global Optimisations for a Prolog Compiler, *Journal of Logic Programming* 2, 1 (April 1985), 43-66.
18. Mellish, C. S., Abstract Interpretation of Prolog Programs, in *Proceedings of the Third Int'l. Conf. on Logic Programming, July, 1985, London, England*, vol. 225, Springer-Verlag, Berlin, 1985, 463-474.
19. Nikhil, R. S., Pingali, K. and Arvind, Id Nouveau, Computation Structures Group Memo 265, MIT Laboratory for Computer Science, July 23, 1986.
20. Nilsson, M., The world's shortest Prolog interpreter?, in *Implementations of PROLOG*, J. A. Campbell (ed.), Ellis Horwood Ltd., Chichester, 1984, 87-92.
21. Reddy, U. S., On the relationship between functional and logic programming, in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, 3-36.
22. Robinson, J. A. and Greene, K. J., New Generation Knowledge Processing/Syracuse University Parallel Expression Reduction, Final report, Syracuse University/Rome Air Development Center, January, 1987.
23. Turner, D. A., A new implementation technique for applicative languages, *Software Practice and Experience* 9, 1 (January 1979), 31-49.
24. Warren, D. H. D., An Abstract Prolog Instruction Set, Tech. note 309, SRI International, October 1983.
25. Wise, M. J., *Prolog Multiprocessors*, Prentice-Hall, 1986.

Section 4

GWAM: A Gadgetized Variant of the Warren Abstract Machine

by

**Tom M. Blenko
Daniel R. Corpron**

Table of Contents

Section	Title	Page
4.1	Introduction	4-1
4.2	Introduction to Gadgets	4-2
4.2.1	Simplified gadget definitions	4-3
4.2.2	Mapping gadgets to the GWAM	4-6
4.3	Definition of GWAM	4-7
4.3.1	Data Structures	4-7
4.3.2	Registers and Variables	4-8
4.3.3	Instruction Set	4-9
4.4	GWAM Implementation	4-21
4.4.1	Data Structure Representation	4-21
4.4.2	Memory Layout	4-21
4.4.3	Instruction Representation	4-22
4.4.4	Instructions Shared by WAM and GWAM	4-22
4.4.5	Allocation Instructions	4-22
4.4.6	Unification Instructions	4-22
4.5	Assessments and Future Work	4-23
4.5.1	Differences between the GWAM and WAM	4-25
4.5.2	Performance characterization	4-26
4.5.3	Support for optimization techniques	4-28
4.5.4	Some indications for future GWAM implementations	4-29
4.6	Acknowledgements	4-30

Table of Figures

Figure	Title	Page
4-1	Gadget for constant 1	4-4
4-2	Gadget for variable X	4-5
4-3	Gadget for list [1]	4-5
4-4	Memory Mapping	4-9
4-5	Instructions shared by WAM and GWAM	4-10
4-6	Comparison of WAM and GWAM Unification Instructions	4-10
4-7	Compiled code for <i>index</i>	4-11
4-8	Compiled code for <i>p1</i>	4-13
4-9	Compiled code for <i>verb</i>	4-13
4-10	Compiled code for <i>family</i>	4-15
4-11	Compiled code for <i>p2</i>	4-17
4-12	Compiled code for <i>family</i>	4-17
4-13	Compiled code for <i>ancestor</i>	4-18
4-14	Compiled code for <i>p3</i>	4-19
4-15	Compiled code for <i>family</i>	4-20
4-16	Memory Mapping	4-22

Table of Tables

Table	Title	Page
4-1	Discrete Unification Actions	4-24
4-2	Benchmark Results	4-27

Section 4

GWAM: A Gadgetized Variant of the Warren Abstract Machine

Abstract

We have designed and implemented the GWAM (Gadgetized Warren Abstract Machine), a variant on the Warren abstract machine. The GWAM incorporates a novel approach to unification, based on previous work in our group in which unification is implemented using higher-order functions termed *gadgets*. The purpose of the GWAM is to provide a state-of-the-art vehicle for demonstrating that technique, and for evaluating powerful optimization methods that become possible with this treatment of unification.

We briefly introduce the principles underlying the gadget technique, and then provide a definition of the GWAM data structures and instruction set. Our implementation of the GWAM is described, with special attention given to differences between it and the WAM implementation from which it is derived. Finally, we discuss properties of the GWAM that make it specially suited for the study of logic-program optimization techniques, and how future implementation efforts might be directed in order to provide a very high level of performance.

4.1 Introduction

Our group has been investigating parallel implementation strategies for combined functional and logic-programming languages. Of particular interest are strategies developed for the parallel implementation of functional programming languages, e.g., combinator reduction [6], and the extensions necessary to support full functional + logic-programming languages. We have developed a novel method for extending functional-programming implementations to support logic-programming languages. This method, termed the *gadget* technique, makes use of higher-order functions and continuation passing, and requires the addition of only three non-functional operations [1].

One interesting property of the gadget technique is that it permits powerful program transformation and optimization methods, previously developed for functional programs, to be applied to logic programs after their translation to almost-functional form. In many cases, these transformations enable compile-time elimination of operations specific to logic programming, and this is of interest from both the implementation and the programming-language perspectives.

The purpose of the work reported here is to provide a state-of-the-art implementation that will serve as a testbed for further development and for evaluation of these transformation techniques. By employing well-understood technology for logic-programming implementation, i.e., the Warren Abstract Machine (WAM) with modifications to support the gadget technique, we provide a vehicle both for identifying performance improvements due to the optimization methods under study, and for comparing absolute performance using our technique with the state of the art.

Prolog has been chosen as the subject language in this effort for several reasons:

- Prolog is a standard language, and both benchmark programs and full-scale applications are available to us.

- The sequential execution of Prolog has been studied in detail, and state-of-the-art implementations are available for comparison purposes.
- The gadget technique for translation of logic programs to functional form is largely independent of the underlying functional programming language implementation. Consequently, there is no penalty for narrowing the focus from functional + logical languages to logic-programming languages alone, or to Prolog in particular; advances at the implementation level are as applicable to translated logic programs as well as functional programs.

This report describes the Gadgetized WAM (GWAM), an abstract machine that employs the gadget technique and is a variant of Warren's Abstract Machine (WAM). The GWAM instruction set

- incorporates gadgets (defined in section 4.2 below) in place of the WAM unification instructions (the relationship of the GWAM to the WAM is discussed in section 4.3).
- provides a suitable target language for translation of logic programs, with or without the optimizing transformations we wish to study.
- can be implemented in much the same fashion as existing WAM implementations.

The remainder of the paper is organized as follows: in section 4.2 we provide an overview of gadgets, the higher-order functions that play a central role in our overall scheme; in section 4.3 we define the GWAM and discuss its relationship to the WAM; section 4.4 discusses our current implementation of the GWAM, including results of a benchmark set; and in section 4.5 we discuss some of the implications of our results for future efforts, including direct compilation of the abstract machine instructions to a conventional architecture, and requirements for a special-purpose architecture.

4.2 Introduction to Gadgets

The intuition behind gadgets is simple: given any two terms, there is a functional encoding for each such that when one is applied to the other, the effect on the program is identical to that achieved by unifying the corresponding terms. So, for example, given two terms t_1 and t_2 and their encodings, signified by $\langle t_1 \rangle$ and $\langle t_2 \rangle$, the function application

$$(\langle t_1 \rangle \langle t_2 \rangle)$$

is to return failure when t_1 and t_2 are non-unifiable, to have no effect if t_1 and t_2 are identical, and so forth. Furthermore, it must be the case that for all t_1 and t_2 ,

$$(\langle t_1 \rangle \langle t_2 \rangle) = (\langle t_2 \rangle \langle t_1 \rangle)$$

In essence, this requires the unification operation to be directly translated to β -reduction, the principal operation underlying functional reduction.

The encodings that achieve this are what we have termed *gadgets*. Their definition is based on modeling unification as a "negotiated" settlement between two parties (the terms), in which a simple or complex term is always willing to signal and relinquish control to its partner. A variable term, on the other hand, will seize control irrespective of its partner and irrespective of whether the partner has previously relinquished control. Each gadget, then, consists of a negotiating algorithm that assures the desired result irrespective of the gadget to which it is applied. As is demonstrated below, this model is readily implemented by a simple coroutining procedure based on continuation passing.

We use the notation of the λ calculus throughout the definitions that follow. While certainly the addition of non-functional operations mean that the language is no longer the λ calculus, the notation makes the role of non-functional operations evident in context. We will also assume a policy of lazy evaluation, or leftmost-first reduction. The importance of functional notation to other properties of the gadget technique, e.g., program transformation, and the role of lazy evaluation in the correctness of gadget definitions are discussed elsewhere [1].

4.2.1 Simplified gadget definitions

We provide some simplified gadget definitions here as an introduction to the fundamental issues addressed by, and methods employed in, the gadget technique. A more detailed account may be found in [1].

Constant gadget

The gadget encoding of the constant term 1 is shown in Figure 4-1. The operation

unify(1,2)

is realized by the function application

(< 1 > continuation < 2 >)

in which < 1 > is a higher-order function, < 2 > is its second argument, and the first argument, *continuation*, represents the continuation for successful unification of 1 with 2.

The gadget definition contains two main branches. The *else* branch of the gadget < 1 > is taken on evaluation of the expression shown above. The value resulting is the function application

(< 2 > continuation % 1)

in which the < 1 > gadget has transferred control to the gadget presented as its argument, < 2 >, with the value it encodes, 1, and a *control token*, %, signaling the control transfer, communicated as arguments. The < 2 > gadget then selects its *then* branch, detects the control token %, and performs an equality check of its encoded value, 2, with the presented value, 1. The value *fail* is then returned. If the gadget in the argument position had initially been < 1 > instead of < 2 >, the equality check would succeed and continue the computation by invoking *continuation*.

This simple coroutining behavior is the heart of the gadget technique.

```

(λN.λA.
  if (A==%)
    then (λV.
      if (V==1)
        then N
        else fail)
    else (A N % 1))

```

Figure 4-1: Gadget for constant 1

Variable gadget

The constant gadget, as shown in Figure 4-1, is always willing to give up control to its partner, and assumes responsibility for the equality check only in the case that its partner has already yielded control. The code for the gadget encoding a logical variable X , shown in Figure 4-2, has similar structure but a different behavior. The variable gadget always assumes control and binds its *canonical representative* to the value presented. The term Xc is the canonical representative of the gadget $\langle X \rangle$ in Figure 4-2.

For example, in the function application

```
( $\langle X \rangle$  continuation  $\langle 1 \rangle$ )
```

the code for $\langle X \rangle$ will detect the absence of a control token and proceed in the *else* branch to determine that $\langle 1 \rangle \neq \langle X \rangle$, and that $Xc == UNBOUND$, with the result that a non-functional *BIND* of $\langle 1 \rangle$ is applied to the term Xc . At the implementation level, *BINDing* can be thought of as form of once-only overwriting, although other implementations are possible. In the case that a control token is present, as in the application

```
( $\langle X \rangle$  continuation % 1)
```

the variable gadget chooses the first *then* branch and inserts the value 1 into a constant gadget (note that this branch builds a constant gadget for the value presented) and then *BINDs* X to it. The binding of a canonical representative is always a gadget.

The other branches in the variable gadget definition handle the case in which the canonical representative X is already bound, so that chain-following is necessary, and the case in which a variable is being unified with itself, a nilpotent operation that simply invokes the continuation.

List gadget

Code for a list gadget (for the list term $[1]$) is shown in Figure 4-3. It is quite similar to the constant gadget, except that the *else* branch prepends the list head and tail to the continuation, with the result that unification of these subterms will be the next order of business. The reader can verify that the application

```

(λN.λZ.
  if (Z==%)
    then if (Xc==UNBOUND)
      then (λB.(BIND Xc
        (λN1.λA.
          if (A==%)
            then (λV.if (V==B)
              then N1
              else fail)
            else (A N1 % B))
          N))
      else (Xc N %)
    else if (Z== < X >)
      then N
    else if (Xc==UNBOUND)
      then (BIND Xc Z N)
    else (Xc N Z))

```

Figure 4-2: Gadget for variable X

(<[1]> continuation <[1]>)

does indeed evaluate to the expression

continuation

as expected.

Gadgets for user-defined structures are handled in much the same way. Both the list and structure gadgets require changes to the variable gadget, so that the variable gadget can handle "reconstruction" of these gadgets in its first *then* branch; the implications are discussed in greater detail in [1].

```

(λN.λA.
  if (A==%)
    then (λV.if (V==cons)
      then (N <first> <second>)
      else fail)
    else (A (λS.λT.(S (T N <second>) <first>))) % 1))

```

Figure 4-3: Gadget for list [1]

4.2.2 Mapping gadgets to the GWAM

The definitions of gadget-specific operations in the GWAM are detailed in section 4.3. It is worthwhile providing an overview here, however, of the (possibly opaque) relationship between gadgets as we have defined them and instructions in a WAM-like instruction set.

Recall from the introduction above that gadgets implement unification, and it is primarily the unification operations of the WAM that we will be replacing. We have split the unification operation into pieces in such a way that one piece is associated with each term, and any two terms can together reconstruct the parts of the unification operation required to unify them. There is no longer any need for an explicit unification procedure.

Term encoding

Every term of a logic program has a type (atom, structure, list, or variable). The WAM represents the type of a term using a tag appended to the address or value comprising the term.

In the gadget view, each term of the logic program has associated with it a higher-order function that, together with an address or value, comprises the gadget for that term. There are four basic gadgets: constant, (user) structure, and list gadgets (also known as nullary, unary, and binary gadgets), and variable gadgets. The gadget encoding a term can therefore use exactly the same data structure as a term in the WAM. The difference lies in the fact that the GWAM tag is interpreted as indexing a higher-order function rather than providing type information to be examined by a monolithic unification routine.

Primitive operations

The three non-functional operations required by the gadget technique are term allocation, *BIND*-ing, and whatever method one uses for maintaining logical variable bindings across backtracking, e.g., trailing with unbinding or copying.

The GWAM uses exactly the same mechanisms as those provided by the WAM for implementing *BIND*ing and backtracking. A different, but similar, approach is taken for doing allocation.

Encoding of control structures

Capturing the coroutining used by gadget definitions, and the prepending of ununified subterms to the current continuation (demonstrated in the list gadget above), are issues for the implementation of gadgets in a WAM-like architecture.

Two observations are essential to addressing this issue. First, the WAM has what can be viewed as rudimentary support for continuations: the current continuation is always the abstract machine program counter, and a copy of the current state is saved at each choice point, so that the program counter stored in the most recent choice point can be popped into the program counter in order to effect backtracking. Second, gadgets as defined are tail-recursive, modulo prepending of ununified subterms to the current continuation. Any state changes induced by execution of a gadget, other than *BIND*ing, are therefore reflected in the arguments it passes to its successor, including, in particular, the continuation argument.

The implementation of gadget control structures in the GWAM reduces to supporting contin-

uations at the relatively fine grain of the unification operation. As is shown in detail in section 4.3, this proves to be strikingly simple. The WAM handles subterm unification by pushing unified subterms onto a local stack. The GWAM uses the same data structure, but instead treats it as a continuation stack, with the program counter inserted as the first element. During unification, this stack always holds the current continuation as its top-of-stack. When the last element of the stack, i.e., the (abstract machine) program counter, is popped, the program counter once again becomes the current continuation.

4.3 Definition of GWAM

This section describes the GWAM and its relationship to the WAM. Much of the WAM data structures, registers, and instruction set have been adopted by the GWAM, so we will focus especially on differences between the two. The gadget technique provides a novel view of the unification operation, so it is not surprising that the greatest differences appear in the abstract machine instructions implementing unification.

The GWAM definition is presented in three parts:

Data Structures

Registers and Variables

Instruction Set

4.3.1 Data Structures

The GWAM consists of three principal data structures normally allocated in main memory

the heap (which is in fact a stack);

the user stack;

the trail stack;

In addition, there is a *local continuation stack* (LCS) that is allocated and freed dynamically, and a *register set* than can be implemented in a variety of ways. A *code area* for storage of user programs may or may not be allocated in main memory.

We will not be concerned with data structures for programs and clauses here: many choices are possible. The logical structure of cells encoding terms of the program, however, is of interest.

The data structure for each term consists of two logical parts: a *value* and a *tag*. There are four types of terms: constants, variables, lists, and structures:

Constants store their value in the value part of the structure and their type in the tag.

Variables store their binding in the value part and their type in the tag. Unbound variables can either be represented by a pointer to themselves in the value part or an unset bound/unbound bit in the tag.

Lists store a pointer to the first element of the list in their value part and their type in the tag part. Our abstract machine assumes that lists are allocated as two adjacent cells in memory, the first containing the head of the list, and the second the tail.

Structures store a pointer to the principal functor in their value part and their type in the tag. Structures are assumed to be allocated in sequential cells in memory, with the principal functor first, the arity N next, and then N structure subterm cells in the order they are found in the term.

The GWAM uses an extended tag part for each cell. The WAM tag encodes only the term type, but the GWAM encodes the identity of the associated gadget. The purpose of our research is to investigate compile-time techniques for optimizing these gadgets, and the extended tag plays an important role in the implementation. For example, the gadget for a formal argument appearing in the head of a clause will always be applied to an actual argument – hence the control token will never be present. The outermost test operation in each of figures 4-1, 4-2, and 4-3 is unnecessary for such terms. Its result is known at compile time, and therefore it is only necessary to execute a special-purpose gadget corresponding to the compile-time-selected branch. Storing this information in the tag permits the GWAM to dispatch to a special-case, rather than general-case, gadget at unification time. It should be evident that only a small number of these special-purpose gadgets need to be provided since there are only a small number of conditionals whose test values may be inferred at compile time.

4.3.2 Registers and Variables

The abstract machine uses several registers to capture the current states of the principal data structures, and to maintain certain control information. These are:

- P* program counter: address of the next GWAM instruction to execute.
- CP* continuation pointer: address of the next instruction to execute should the current goal succeed.
- B* backtrack pointer: address of the last choice point pushed on the user stack.
- E* environment pointer: address of the last environment pushed on the user stack.
- TR* trail pointer: address of the top of the trail.
- H* heap pointer: address of the top of the heap.
- HB* heap backtrack pointer: the value of *H* at the time of the last choice point was placed on the stack.
- FA* formal argument pointer: address of formal argument (a structure) being unified.
- AA* actual argument pointer: address of actual argument being unified when formal argument is a structure.
- $A_1, A_2, \dots$ argument registers: set to actual arguments prior to executing the current goal.
- $X_1, X_2, \dots$ temporary variables: A_i and X_i registers are identical: the different names reflect different usages. Temporary variables are variables that can be implemented as

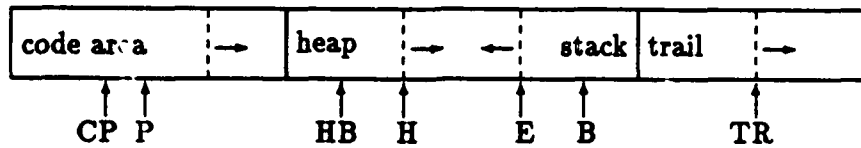


Figure 4-4: Memory Mapping

registers because their scope is limited to the procedure call in which they are allocated. All other variables are *permanent* variables.

Figure 4-4 shows one possible allocation of GWAM data structures and register assignments in main memory.

4.3.3 Instruction Set

Like the WAM, the GWAM has five classes of instructions. The three classes listed in Figure 4-5,

- Clause indexing
- Clause control
- Clause body

are identical in the two abstract machines.

The instructions that implement unification are constructed rather differently in the two models, as shown in Figure 4-6.

Clause Indexing

Clause indexing instructions link together the clauses that compose a Prolog procedure. They act as an initial filter on the set of candidate clauses, to select those that may be unifiable with the actual arguments.

switchonterm A_i, L_1, L_2 — is usually the first instruction in code for Prolog procedures having more than one clause. An actual argument is already in the A_i register. **switchonterm** branches to an appropriate label depending on the type of A_i .

L_1 labels a block of code to execute if A_i is a constant (including integers and floating point values). L_2 labels a block of code to execute if A_i is a list or structure. If A_i is a variable, all clauses must be attempted, so execution simply falls through to the next statement.

Specialized indexing instructions may be added to optimize procedures with either constants, structures or lists in their heads.

try L, N — is the instruction for the first of several clauses to be attempted. A choice point is allocated on the stack to save all the environment registers, including the N argument registers

Class	Instructions
Clause Indexing	switchonterm try retry trust try_me_else retry_me_else trust_me_else_fail
Clause Control	allocate execute call deallocate proceed
Clause Body Instructions	puttvar putpvar putpval putuval putcon putnil putstr putlist

Figure 4-5: Instructions shared by WAM and GWAM

WAM Class	Instruction	Instruction	GWAM Class
Clause Head Instructions	get_variable get_value get_constant get_nil get_structure get_list	bldpvar bldtvar bldpval bldtval bldcon bldnil bldstr_gadget bldlst_gadget	Allocation Instructions
Generalized Unification	unify_pvar unify_tvar unify_pval unify_con unify_nil	pvar_gadget lst_gadget str_gadget pval_gadget val_gadget con_gadget nil_gadget	Unification Instructions

Figure 4-6: Comparison of WAM and GWAM Unification Instructions

index:	switchonterm A_1, S_1, S_2
	try S_1
	trust S_2
S_1 :	code for clause 1 which has a constant as its first formal argument
	$\vdots$
S_2 :	code for clause 2 which has a list as its first formal argument
	$\vdots$

Figure 4-7: Compiled code for index

(N = procedure's arity). A pointer to the code following this instruction is also saved in the choice point for resumption in case of failure. Execution proceeds with the clause at label L .

retry L, N — is used for each "middle" clause among several to be attempted. The N argument registers are restored from the choice point allocated by try and execution proceeds with the clause at L . A pointer to the code following this instruction is stored for resumption on failure.

trust L, N — is used for the last clause among several being attempted. Since the clause is the last to be attempted, it is "trusted" to succeed, otherwise the parent goal will fail. N argument registers are restored from the choice point previously allocated by try and the choice point is discarded. Execution continues at L .

try_me_else L, N — is similar to the try instruction except that execution is continued immediately following the instruction and the label L is stored as the location to resume upon failure. Like try, it creates a choice point on the stack and saves all the environment registers.

retry_me_else L, N — is like retry except that execution continues with the following instruction, and the choice point is updated by storing L as the location for resumption upon failure.

trust_me_else_fail N — is like trust except that execution continues with the immediately following instruction.

Example

As an example, consider the compiled code for the following Prolog procedure as shown in Figure 4-7:

index(c,X) :- ...	(1)
index([a,b],X) :- ...	(2)

Suppose the value of A_1 is a constant. Then clause 1 is the only one whose head could possibly unify with it; consequently the switchonterm instruction branches to label S_1 . Similarly, if A_1 is a list, only clause 2 is tried. If A_1 dereferences to a variable, control falls through to the try and trust instructions, which cause both clauses to be attempted.

Clause Control

Clause control instructions are responsible for the control transfer and environment maintenance associated with execution of a single clause. They arrange control transfer to subgoals under varying circumstances, and allocate and deallocate space from the user stack.

allocate — appears at the beginning of code generated for a single clause containing more than one subgoal and requiring space for permanent variables. It allocates an environment on the top of the user stack.

execute P — is generated as the last instruction in the code for a clause body. It implements tail-recursive transfer, executing a goto statement with destination P . Its use is demonstrated in the example below.

call P, N — invokes a subgoal P with arity N (other than the last one) in the body of a clause. A pointer to the remainder of the clause body is stored, and control is transferred to the code for P . Its use is demonstrated in the example below.

deallocate — deallocates an environment from the stack when it is no longer needed, before the final execute instruction in the code for the clause body. The current environment is discarded and the continuation stored by the parent is restored to the register CP .

proceed — terminates a unit clause. Unification with the unit clause has completed successfully and therefore the parent goal has succeeded as well. Control is transferred to the continuation stored by the parent.

Examples

The compiled code (Figure 4-8) for the following procedure demonstrates several of the clause control instructions:

```
p1(X,Y,Z) :- q1(Y),q2(X,Y),q3(Z).
```

The role of proceed is demonstrated by the compiled code in Figure 4-9 for the procedure

```
verb(run).  
verb(catch).  
verb(fly).
```

Clause Body Instructions

Clause body instructions are identical to the WAM put instructions. They move data to the argument registers ($A_1, A_2, \dots$) immediately prior to calls to subgoals in a clause body. No unification is performed. These instructions take the form `putxxx Source, Dest`, where `Dest` is some subgoal's argument register and `Source` is some variable (permanent or temporary) or a constant (including integers and floating point).

```

p1:  allocate
      code for executing the variable gadgets X, Y, and Z
      :
      code for preparing the actual argument gadget for subgoal q1
      call q1,1
      :
      code for preparing the actual argument gadgets for subgoal q2
      call q2,2
      :
      code for preparing the actual argument gadget for subgoal q3
      :
      deallocate
      execute q1

```

Figure 4-8: Compiled code for p1

```

verb:  try verb1,1
        retry verb2,1
        trust verb3,1
verb1:  code for verb(run)
        :
        proceed
verb2:  code for verb(catch)
        :
        proceed
verb3:  code for verb(fly)
        :
        proceed

```

Figure 4-9: Compiled code for verb

puttvar X_i, A_j — is used for temporary variables that appear as arguments of goals in a clause body. An unbound variable is pushed onto the heap with X_i and A_j both pointing to it.

putpvar Y_i, A_j — is used for permanent variables that appear as arguments of goals in a clause body. It simply binds the argument register A_j to Y_i .

putpval Y_i, A_j — is used in situations where the permanent variable pointed to by Y_i which may already be bound. A_j is bound to the value stored in Y_i or its dereferenced value.

putuval Y_i, A_j — is used when Y_i is "unsafe".

The code interpreter employs a generalized form of tail recursion optimization by deallocating the environment of the current clause before the final goal in a clause body is executed. If an argument to the last goal contains references to an unbound variable in the current environment, deallocation might result in dangling references. A variable is *unsafe* in a situation where deallocation of its environment might produce dangling references.

putuval avoids this problem by examining the unsafe variable Y_i at run time to see if it dereferences to a variable in the current environment. If so, that variable is bound to a new global variable created on the heap (making it safe) and the actual argument is set to point to it. Otherwise, the dereferenced value of Y_i , rather than Y_i , is copied to the argument register.

putcon *Const*, A_i — puts a copy of the named constant in register A_i .

putnil A_i — is a special-case instruction semantically equivalent to **putcon** [], A_i .

putstr *Struct*, A_i — marks the beginning of a structure. The functor *Struct* is pushed onto the heap, and register A_i is set to point to it. The structure's subterms are constructed by the **bld** instructions discussed below.

Abstract machine code for the procedure

$\text{family}(\text{ancestor}(X,Y)) \text{ :- family}(\text{parent}(X,Y)).$ (3)

$\text{family}(\text{ancestor}(X,Y)) \text{ :- family}(\text{parent}(X,Z)), \text{family}(\text{ancestor}(Z,Y)).$ (4)

is listed in Figure 4-10 and provides an example of the use of **putstr**.

putlist A_i — marks the beginning of a list. It puts a list pointer pointing to the top of heap in register A_i . The elements of the list are constructed by the **bld** instructions discussed below.

Allocation Instructions

Allocation instructions dynamically allocate list or structure subterms occurring in the head or body of a clause.

A list or structure term is initialized on the heap by one of several instructions (**putlist** or **putstr**, defined above, or **bldstr_gadget** or **bldlst_gadget** defined below). The **bldxxx** instructions then allocate all subterms in immediately-following locations on the heap.

Other than where noted below, the term *structure* will refer to both lists (e.g. [a,b,c]) and principle functor structures (e.g. *father*(abraham,isaac)).

```

family:  try family1,1
         trust family2,1
family1: instructions to allocate the ancestor structure on the heap
         :
         instruction to unify the actual and formal argument
         putstr (parent,2),A1
         instructions to allocate the structure parameters on the heap
         :
         execute family
family2: allocate
family1: instructions to allocate the ancestor structure on the heap
         :
         instruction to unify the actual and formal argument
         putstr (parent,2),A1
         instructions to allocate the structure parameters on the heap
         :
         call family,2
         putstr (ancestor,2),A1
         instructions to allocate the structure parameters on the heap
         :
         deallocate
         execute family

```

Figure 4-10: Compiled code for family

bldlst_gadget A_i — is used when a list term occurs as the i^{th} formal argument in the head of a clause. The *FA* register is set to top-of-heap (where the elements of the list will subsequently be allocated), and A_i is copied to the *AA* register.

bldstr_gadget *Struct*, A_i — is used when a structure term occurs as the i^{th} formal argument in the head of a clause. The functor *Struct* is copied to a new cell allocated from the heap (where the subterms of the structure will subsequently be allocated), *FA* is set to point to it, and A_i is copied to the *AA* register.

bldpvar Y_i — allocates a permanent variable that occurs as a subterm in a structure. A variable is known to be unbound only in its first occurrence (in left to right order) in a clause. A new, unbound variable is allocated on the heap and the register Y_i is set to point to it.

bldtvar X_i — works like **bldpvar** except that the argument is a temporary rather than a permanent variable. A new, unbound variable is allocated on the heap and the register X_i is set to point to it.

bldpval Y_i — is used for possibly-bound, permanent variables that appear as subterms of a structure. A variable may be bound in any but its first appearance in a clause. **bldpval** pushes the (dereferenced) contents of Y_i onto the heap.

bldtval X_i — works like **bldpval** except its argument is a temporary rather than a permanent variable. The value of the temporary variable X_i is pushed onto the heap if it is bound, otherwise a new variable is allocated on the heap.

bldcon *Const* — is used for constants that appear as subterms in structures. *Const* is pushed onto the heap.

bldnil — is equivalent to **bldcon []**.

Allocation instructions implement a subset of the functionality provided by WAM unify instructions. In particular, the instructions that initiate structure unification (**get_structure**, **get_list**, **putstr**, **putlist**) in the WAM also set a read/write register that is examined by subsequent instructions. WAM unify instructions are then used for structure subterms. In their write mode, the unify instructions perform the same operations as corresponding GWAM **bldxxx** instructions.

Examples

As an example of list allocation examine the abstract machine code in Figure 4-11 generated for the following Prolog clause in which X is a temporary variable and Y is a permanent variable:

$\text{p2}(X,Y,[X|Y]) \text{ :- } q(Y,X),r(Y).$

As another example of the use of the allocation instructions, Figure 4-12 shows in greater detail the compiled code for clause 3 and clause 4 listed above.

```

p2:  allocate
      movreg A1,X4
      pvar_gadget Y1,A2
      bldlst_gadget A3
      bldtval X4
      bldpval Y1
      instruction to unify the actual and formal argument
      putpval Y1,A1
      movreg X4,A2
      call q,2
      putpval Y1,A1
      deallocate
      execute r

```

Figure 4-11: Compiled code for p2

```

family:  try family1,1
          trust family2,1
family1:  bldstr_gadget (ancestor,2), A1
          bldtvar X3
          bldtvar X4
          instruction to unify the actual and formal argument
          putstr (parent,2),A1
          bldtval X3
          bldtval X4
          execute family
family2:  allocate
          bldstr_gadget (ancestor,2), A1
          bldtvar X3
          bldpvar Y1
          instruction to unify the actual and formal argument
          putstr (parent,2),A1
          bldtval X3
          bldpvar Y2
          call family,2
          putstr (ancestor,2),A1
          bldpval Y2
          bldpval Y1
          deallocate
          execute family

```

Figure 4-12: Compiled code for family

```

ancestor:  con_gadget joe, A1
           con_gadget mary, A2
           code to prepare the argument gadgets for the clause body
           :

```

Figure 4-13: Compiled code for ancestor

Unification Instructions

Unification instructions largely replace WAM `get` and `unify` instructions. For each formal argument, which is allocated by a sequence of `bldxxx` instructions, there is a single unification instruction with a name of the form `xxx_gadget`. Each unification instruction encodes the type of the corresponding formal argument, and takes the formal argument location and the procedure's actual argument as its arguments. This scheme could be generalized to a single, all-purpose unification instruction that dispatched on the tag of the formal argument (rather than encoding it).

The unification instructions initiate the actions dictated by the gadget definitions for the terms being unified. For example, given the procedure

ancestor(joe,mary) :- ... (5)

in its compiled code (shown in Figure 4-13), no allocation (of constants) is necessary. The `con_gadget` instruction causes code for a constant gadget (shared by all constants other than `nil`) to be invoked using `joe` as its constant. Upon completion, it dispatches on the tag of the second argument, `mary`, which happens to be a constant as well, and executes the code for its associated gadget. The implementation-level code for the constant gadget, then, is executed twice: once each for the terms `joe` and `mary`.

We have divided the unification instructions into two groups: simple and structural. Simple instructions encode unification of formal arguments that are constant, integer, floating point, and variable (bound and unbound) terms. Structural instructions encode unification of list and structure terms.

Simple Unification Instructions

Simple unification instructions handle all constant (integers and floating point numbers included) and variable (permanent and temporary, bound and unbound) terms that appear as formal arguments of a clause.

`con_gadget Const, Ai` — unifies a constant formal argument, *Const*, with the i^{th} formal argument to the clause. If unification is successful, execution continues with the next abstract machine instruction. Otherwise, the goal fails and backtracking occurs.

`nil_gadget Ai` — represents a special case of `con_gadget`. It is equivalent to `con_gadget [], Ai`.

`pvar_gadget Yi, Aj` — binds the permanent variable gadget *Y_i* to the dereferenced value of the actual argument gadget *A_j*. Note that *Y_i* is guaranteed to be unbound.

```

p3:  allocate
      int_gadget 2, A1
      pvar_gadget Y1, A3
      movreg A2, A1
      call q1,3
      putpval Y1,A1
      deallocate
      execute q2

```

Figure 4-14: Compiled code for p3

Figure 4-14 illustrates compilation of the following code which contains a temporary variable X , allocated in register X_2 , and a permanent variable Y .

$p3(2,X,Y) :- q1(X), q2(Y).$

Note that the `movreg` instruction suffices to move the value of the temporary variable X to the register in which it becomes an argument in the call to procedure `q1`.

val_gadget X_i, A_j — unifies the formal argument, a temporary variable allocated in register X_i , with the actual argument presented in A_j . This instruction is used for formal arguments represented by temporary variables that have occurred previously in the clause head and may already have been bound.

pval_gadget Y_i, A_j — unifies the permanent variable Y_i with the actual argument gadget A_j . This instruction is used for formal arguments represented by permanent variables that have already occurred that have occurred previously in the clause head and may already be bound.

Structural Unification Instructions

Structural unification instructions handle all lists and structures that appear as a formal arguments of a clause. Their two arguments are implicit, the registers FA and AA , which contain the formal and actual arguments, respectively, and are initialized by either the `bldlst_gadget` or `bldstr_gadget` instructions.

lst_gadget — unifies a formal argument (a list) with an actual argument. The list term, built by `immediatei`, preceding `bldxxx` instructions and pointed to by the FA register, is unified with the actual argument, which is pointed to by the AA register. Unification subproblems are pushed onto the Local Continuation Stack, to be executed if unification at the outermost level is successful. Otherwise, the current goal fails and backtracking is initiated.

str_gadget — works like `lst_gadget`, except that FA points to a structure term on the heap built by immediately preceding `bldxxx` instructions.

Figure 4-15 shows the entire abstract machine code for clauses clause 3 and clause 4 listed above and demonstrates the use of `bldstr_gadget` and `str_gadget`.

```

family:  try family1,1
         trust family2,1
family1: bldstr_gadget (ancestor,2), A1
         bldtvar X3
         bldtvar X4
         str_gadget
         putstr (parent,2),A1
         bldtval X3
         bldtval X4
         execute family
family2: allocate
         bldstr_gadget (ancestor,2), A1
         bldtvar X3
         bldpvar Y1
         str_gadget
         putstr (parent,2),A1
         bldtval X3
         bldpvar Y2
         call family,2
         putstr (ancestor,2),A1
         bldpval Y2
         bldpval Y1
         deallocate
         execute family

```

Figure 4-15: Compiled code for family

4.4 GWAM Implementation

Our implementation of the GWAM is based on SB-Prolog, a Prolog implementation developed at SUNY at Stony Brook and the University of Arizona. The implementation is known to run on Sun microcomputers, and may run on Vaxes or other Unix-based systems as well.

For the most part, we have adopted commitments made by that implementation in the areas of

- The Prolog language variant supported.
- Choice of concrete data structures in the implementation.
- Memory layout for principal data structures.
- Features such as tail-recursion optimization, static procedure declaration and optimization, etc.

The user manual [3] and code can be consulted for further details.

Like most WAM implementations, SB-Prolog is based on environment stacking rather than goal stacking.

The SB-Prolog implementation consists of a monolithic abstract machine interpreter in which a single switch statement provides dispatch for the current instruction in the input stream. This distinguishes it from the Quintus implementation, which compiles to threaded code and requires only a threaded code interpreter.

4.4.1 Data Structure Representation

The concrete data structures representing procedures and clauses are not of interest to us here; the GWAM structures are identical to the SB-Prolog structures, and their details can be gleaned from the source code.

The data structure cell representing a Prolog term is embedded in a 32 bit word. The logical structure of these has been discussed earlier in Section 4.3.1. Since all structures are allocated on word boundaries, the two least significant bits are available for the tag, and the remainder are used for the value. The current implementation does not include extended tags, but our expectation is that 6 or 8 bits taken from the most significant end of the word will suffice.

4.4.2 Memory Layout

The memory layout of principal data structures in SB-Prolog is depicted in Figure 4-16. The relative positions of the stack and the heap in the address space are essential to certain operations, and dangling references are avoided by always binding the variable with the lowest address when unifying two variables.

All registers are allocated in low memory at compile time, and instances of the *LCS* are allocated (and freed) dynamically from free memory.

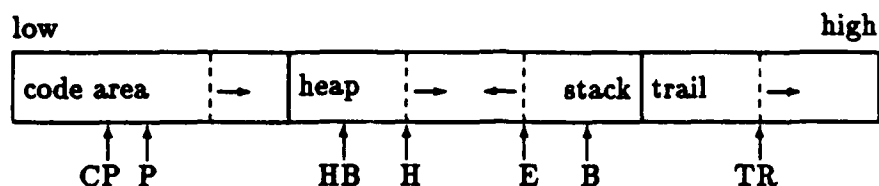


Figure 4-16: Memory Mapping

4.4.3 Instruction Representation

Each GWAM instruction is encoded by a single byte and followed immediately in the code area by its operands. The total space occupied by an instruction's operands depends on how many there are (0 to 3), and whether they represent constant values, register numbers, or indices to variables in the local environment.

4.4.4 Instructions Shared by WAM and GWAM

The interpreter implements the operations described in Section 4.3 for each clause indexing, clause control, and clause body instruction. The code for these is unchanged from the initial SB-Prolog implementation. Since their implementation has been discussed elsewhere [7,3,4,5], we will only briefly discuss some relevant implementation details here.

4.4.5 Allocation Instructions

With the exception of `bldlst_gadget` and `bldstr_gadget` each allocation instruction implements the write mode version of the corresponding WAM unify instructions. All allocation is from the top of the heap (which is used as a stack, but called a heap for historical reasons).

In addition to allocation, `bldlst_gadget` and `bldstr_gadget` each copy their operand to the *AA* register and initialize the *FA* to point to top of stack. These two special purpose registers are referenced implicitly by the `lst_gadget` and `str_gadget` unification instructions.

4.4.6 Unification Instructions

As discussed in Section 4.2, the gadget for a formal argument is always applied to the gadget for an actual argument. Therefore, the outcome of the outermost test operation in each formal argument gadget (figures 4-1, 4-2, and 4-3), which tests for presence of a control token, is always known at compile time, i.e., the `else` branch always selected. Our choice of type-specific unification instructions, and their implementation, reflect this knowledge: the instructions act as special-purpose gadgets, performing only the operations of the selected `else` branches.

The GWAM unification instructions and our implementation also adopt optimization on free variables from the WAM. Whenever a variable is known to be unbound, the special-purpose `pvar_gadget` instruction is used; if the binding status is unknown, the more general `pval_gadget` instruction, which checks the binding status first, is employed.

Another optimization in the current implementation is to pass the type of the calling gadget to the called gadget as part of the control token. This allows the called gadget to return failure immediately if the control token supplied signifies a type it cannot match. The control token transmitted by each calling gadget type is summarized by the table:

<i>Gadget type</i>	<i>Control Token</i>
constant	constant
list	list
structure	structure
variable	<i>not applicable</i>

In our current implementation, three registers are used to execute each unification instruction:

a *CTR* (control token register), which stores the state of the control token across calls to gadget code and has a very brief lifetime;

a *Value* register, which contains the value (e.g., a constant) for the type of gadget being executed, and whose tag has dictated selection of the gadget code being executed;

an *Arg* register, which contains a (gadget) term signifying the term to be unified with (when no control token is present), or a value passed from the just-executed gadget (when a control token is present).

These registers can be thought of as implementing arguments in the function calls that are used in defining gadget theory (see Section 4.2). *lst_gadget* and *str_gadget* copy *FA* and *AA* to *Value* and *Arg* respectively and then use these registers implicitly.

Figure 4-1 summarizes the actions implemented by code for each of the four gadget types on receipt of the control tokens indicated, or in the absence of a token.

As an example, when the interpreter encounters the instruction *con_gadget mary, A₂*, it makes the following assignments:

Value = *mary*
CTR = constant
Arg = *A₂*

This achieves the effect of the outermost *else* branch of the constant gadget (see Figure 4-1).

Dispatch on the tag of *A₂*, a logical variable, to code for the logical variable gadget is undertaken directly. Examination of Figure 4-1 shows that the specified action is to bind *Arg* (the register *A₂*) to *Value* (the constant *mary*) if *A₂* is unbound, and this coincides with our expectations for correctness.

4.5 Assessments and Future Work

The GWAM definition and implementation has been undertaken as a vehicle for supporting gadget code in the context of a state-of-the-art implementation of Prolog. In this section we will summarize our conclusions from the work performed, and indicate how it relates to the next stage: evaluation of compile-time optimization techniques adopted from the functional-programming field.

Gadget Type	CTR	Action
constant	constant	Test for equality of constants in <i>Arg</i> and <i>Value</i>
	absent	Interchange <i>Arg</i> and <i>Value</i> , set <i>CTR</i> = constant, continue with current continuation.
	list	Unification fails
	structure	Unification fails
list	list	Push tails, then heads, of lists in <i>Arg</i> and <i>Value</i> onto the LCS
	absent	Interchange <i>Arg</i> and <i>Value</i> , set <i>CTR</i> = list, and continue with current continuation.
	constant	Unification fails
	structure	Unification fails
structure	structure	Test for equality of principal functors of the two structures. If equal, push the parameters of both <i>Arg</i> and <i>Value</i> in pairs onto the LCS.
	absent	Interchange <i>Arg</i> and <i>Value</i> , set <i>CTR</i> = structure, and continue with current continuation.
	constant	Unification fails
	list	Unification fails
variable	anything	If the variable is unbound, bind <i>Arg</i> to <i>Value</i> . Otherwise, dereference <i>Arg</i> and continue with current continuation

Table 4-1: Discrete Unification Actions

4.5.1 Differences between the GWAM and WAM

The chief influence of the gadget technique on logic-programming implementation has been to change the treatment of unification adopted in the implementation. As mentioned earlier, several data structures from the WAM have been preserved in the GWAM but put to different use (e.g., the Local Continuation Stack). These are of some interest, since they reflect a transition to a continuation-based view of the implementation, but we instead focus on more notable differences, which will be discussed in the following subsections.

Elimination of monolithic unification

The conventional approach is to treat unification as a part of the procedure-call mechanism, so that a monolithic unification procedure is called at run-time to perform the desired operation. The WAM supports a somewhat more sophisticated view than that captured by the monolithic unification model, by providing special-purpose unification instructions: code is generated to unify formal arguments available at compile time using these more efficient instructions. Arguments which are instantiated dynamically, however, must still be handled using the general-purpose unification procedure.

Gadgets and the GWAM support a different view: there is no monolithic unification routine, and a piece of the unification procedure, a higher-order function termed a gadget, can be associated with each term in such a way that any pairing of the gadgets suffices to unify the corresponding terms.

Our future work on optimization techniques depends heavily on this property of the GWAM (see section 4.5.3 below).

Allocation of formal arguments

Since the WAM decomposes formal arguments for the purpose of identifying the appropriate special-purpose instructions to implement their unification, it is also advantageous to combine the allocation of (a piece of) a formal argument with its unification in a single instruction. This, for the most part, is what the WAM designers have done. The result is a 'lazy' style of allocation. One piece of a formal argument is allocated and unified: if that succeeds, the next follows. But if unification fails, as it frequently does in Prolog programs, allocation of ununified (because of failure) formal arguments never takes place.

The GWAM does not decompose a formal argument: it allocates all of it and then initiates unification. The unification of the total formal/actual argument pair is represented by a single abstract machine instruction. The abstract machine code for a procedure, then, consists of allocation instructions for a complete formal argument, followed by the (single) unification instruction, followed, possibly, by additional allocation/unification pairs.

This difference between the two abstract machines has negative implications for performance of the current implementation (discussed in Section 4.5.2 below) and positive implications for future implementations (discussed in Section 4.5.4 below).

4.5.2 Performance characterization

The programs used to benchmark our implementation are from a set developed by F. Pereira. These are based on the ICOT benchmark set, and have been used previously in our group as part of other evaluations [2].

Each benchmark was designed to demonstrate a distinct aspect of a Prolog implementation (e.g., list unification). Although they permit accurate comparison of specific aspects of the WAM vs. GWAM in the current implementation, they have little predictive value for the performance of "typical" Prolog programs.

Table 4-2 provides information about the benchmarks and their execution times.

The columns labeled *Formal args.* and *Actual args.* illustrate the correspondence between formal and actual argument types. For example, benchmark 2-1, *deep backtracking* is a program with two parameters, the first a variable and the second a constant. Also, it is invoked with two variables as actual arguments.

The timing information for each benchmark was derived from its elapsed execution time (measured in microseconds) and the number of iterations (up to 2000).

The Δ Column shows the difference in time ($\Delta = GWAM - WAM$), and $\% \Delta$ gives the percentage change between the two implementations ($\% \Delta = \Delta / WAM$).

The benchmarks have been sorted into three groups, marked 1-1 through 1-9 (GWAM slower than WAM), 2-1 through 2-8 (GWAM indistinguishable from WAM), and 3-1 through 3-13 (GWAM faster than WAM).

Benchmarks 1-1 through 1-3 and 1-7 through 1-8 show a marked slowdown for the GWAM relative to the WAM. We have not done detailed profiling, but believe the performance penalty is mostly chargeable to differences in formal argument allocation between the two machines.

The 'unchanged' group is classified as such because the performance differences observed are relatively small and of comparable magnitude to variances observed between runs of the same benchmark. We also have no reason for expecting the Δ 's for 1-9 and 2-1, or for 2-5 and 3-13, to differ, since the mechanisms being tested are unmodified in the GWAM implementation. Consequently, the observed differences may be chargeable to experimental variance, and in any case appear small enough to be considered inconsequential for our purposes.

Benchmarks 3-1 through 3-13 show a modest speedup for the GWAM relative to the WAM. This appears to be due to two factors. First, while the code for gadgets in functional form calls for two procedure calls for each level of unification in a term, and this would normally produce a performance penalty, procedure calls between gadgets are implemented in registers (Value, CTR, and Arg) and are very fast. Second, since every step of the unification is done by code specialized especially for the terms at hand, i.e., their associated gadgets, we expect unification to run somewhat faster than when more general code is run. This is especially demonstrated by those benchmarks in which the formal arguments are variables (hence their gadgets are invoked first), and the actual arguments are structures (hence the variable gadget simply binds the variable to the actual argument). It should be noted, of course, that the performance advantages nevertheless are modest in all cases.

Benchmark	Formal args.	Actual args.	WAM $\mu\text{sec}/\text{iter}$	GWAM $\mu\text{sec}/\text{iter}$	Δ	% Δ
1-1. list traversal*	L	L	3.84	7.46	3.62	94.27
1-2. rec. list traversal*	L	L	1.47	1.83	0.36	24.49
1-3. rec. traversal of 2 lists*	L	L	2.63	3.09	0.46	17.49
1-4. rec. traversal of 4 lists*	L	L	4.60	4.96	0.36	7.83
1-5. rec. traversal of 8 lists*	L	L	8.02	8.82	0.80	9.98
1-6. rec. traversal of 16 lists*	L	L	15.35	16.58	1.23	8.01
1-7. structure traversal†	S	S	3.85	10.09	6.24	162.08
1-8. rec. structure traversal†	S	S	3.20	11.48	8.28	258.75
1-9. shallow backtracking	V,C	V,C	3.96	4.26	0.30	7.58
2-1. deep backtracking	V,C	V,V	7.40	7.62	0.22	2.97
2-2. assert unit clauses	S	S	11586.67	11355.00	-231.67	-2.00
2-3. access unit clauses	V,V	C,C	449.13	438.65	-10.48	-2.33
2-4. slow access of unit clauses	V,V	C,C	646.67	645.00	-1.67	-0.26
2-5. setof	NA	NA	785.33	776.50	-8.83	-1.12
2-6. pair setof	NA	NA	1037.33	1052.00	14.67	1.41
2-7. double setof	NA	NA	2489.33	2475.00	-14.33	-0.58
2-8. floating add	V,V,V	C,C,V	85.76	84.12	-1.64	-1.91
3-1. list construction*	L	V	4.11	3.71	-0.40	-9.73
3-2. structure construction†	S	V	4.34	3.91	-0.43	-9.91
3-3. choice point	C	C	4.90	4.49	-0.41	-8.37
3-4. trail variables	V	C	6.88	6.13	-0.75	-10.90
3-5. medium unify	V,V	L,L	4.69	4.41	-0.28	-5.97
3-6. deep unify	V,V	L,L	300.93	282.07	-18.86	-6.27
3-7. extract 1 st arg.	V,V,V	C,S,V	6.08	5.26	-0.82	-13.49
3-8. extract 2 nd arg.	V,V,V	C,S,V	6.06	5.16	-0.90	-14.85
3-9. extract 4 th arg.	V,V,V	C,S,V	6.05	5.18	-0.87	-14.38
3-10. extract 8 th arg.	V,V,V	C,S,V	5.95	5.18	-0.77	-12.94
3-11. extract 16 th arg.	V,V,V	C,S,V	6.14	5.19	-0.95	-15.47
3-12. integer add	V,V,V	C,C,V	4.55	3.72	-0.83	-18.24
3-13. bagof	NA	NA	327.33	312.50	-14.83	-4.53

C = constant
 V = variable
 L = list
 S = structure
 NA = not applicable

Table 4-2: Benchmark Results

*Benchmark operates on flat lists of 100 elements.

†Benchmark operates on structures with two parameters nested 100 deep, for example
 $f(1, f(2, f(3, \dots f(99, f(100, \text{nil})) \dots))$.

4.5.3 Support for optimization techniques

One of the very attractive properties of gadget-based implementations is the ease with which it appears they will support compile-time optimizations, and of course we intend to study this further. An entirely unanticipated benefit, which we discuss below, is that in implementations such as our current one, implementation can be very compact – and indeed there is little or no cost associated with adding additional special-purpose gadgets!

Extended tags

Extended tags play an important role in our plans for exploring compile-time optimizations. The question we will be answering is, “What information can/should be captured by extended tags.” This is equivalent to asking what special-purpose gadgets we might define in a functional implementation that takes advantage of the same optimizations.

There are a number of candidates (for information to be stored in the tags) that we have already seen:

variable not bound (variable gadgets) – if the variable is known to be unbound (first occurrence in code), one test can be eliminated. The WAM, and the current version of the GWAM (where possible), use a special-purpose instruction to capture this.

variable is bound (variable gadgets) – if the variable is known to be bound, one test can be eliminated. We expect partial evaluation to eliminate such code altogether as our optimization techniques become more sophisticated. The WAM and GWAM capture this to a limited extent through use of temporary variables.

control token not present (all gadgets) – when the gadget in function position knows that its argument is not a control token, one test is eliminated. For example, all formal argument gadgets occupy the functional position. The GWAM `xxx_gadget` instructions already perform this optimization.

control token is present and of type X (variable gadgets) – if a unification operation can be partially reduced (at compile time) to an expression in which a control token is present, we need to be able to express that fact. The GWAM currently sets the control token only in the CTR, and such control tokens are short-lived. This must be done differently if the optimization is to be made available, and what we have in mind is to set the control token as part of the tag of the gadget in argument position.

The first two of these correspond to known optimizations incorporated by the WAM and the GWAM. The second two apply only to the GWAM, and the first is presently implemented in a different fashion.

As mentioned earlier, encoding of this information in the tag, and interpretation as part of dispatch on the tag, would allow the GWAM to revert to a single unification instruction.

There must be a theory of tag copying/propagation developed for the extended tags, of course.

Reentrant property of code implementing gadgets

A very elegant property of our implementation is that all the special-purpose gadgets encoded in the abstract machine implementation are sub-cases of the four basic gadgets (variable, constant, structure, and list). Encoding them proves to be very simple – an additional special-purpose gadget requires addition of one entry to the dispatch table and a single entry point (label) to the existing code. So, for example, the abstract machine code

`con_gadget mary, A2`

reflects the information available at compile time that `mary` is a constant and a formal argument. It therefore branches directly to the `else` branch of the code for constant gadgets. This allows a very compact implementation, in which operations shared by different special-purpose versions of the same general-purpose gadget actually share code in the implementation where appropriate, and the marginal cost for supporting more refined gadgets is very small.

4.5.4 Some indications for future GWAM implementations

Our experience with the current implementation has suggested three possible directions in which research into future gadget-based implementations might be taken.

Direct compilation

Several existing WAM implementations compile Prolog code to byte code encoding WAM instructions. The byte code is then interpreted by a byte-code interpreter. This strategy provides for more compact code than a direct-compiled implementation, and can be expected to reduce the cost of porting an implementation to a new host.

The partitioning of allocation instructions and unification instructions, which is advantageous in the GWAM but less so in the WAM, suggests that it would be very desirable in the GWAM to eliminate most or all run-time allocation of formal arguments altogether. This would require a scheme in which a GWAM procedure had associated with it a data segment in which the formal arguments had been previously allocated. Of course, it is essential to make this area shareable (to support recursion), so a scheme based on structure-sharing would need to be devised. Such a scheme, together with reentrancy of gadgets and the associated compactness of their implementation, could well make direct compilation practical from the point of view of memory requirements, and extremely attractive in terms of performance.

Requirements for a special-purpose architecture

Both the benefits of providing support for extended tags, and the compactness attainable in the implementation of code for gadget instructions, suggest advantages to be obtained from hardware support for some variant of the GWAM.

Interpretation of, and dispatch on, extended tags should be run very fast in hardware, and could proceed in parallel with other operations. This is a strategy exploited in the implementation of custom hardware for Lisp machines.

The compactness of unification code, which is used heavily in the execution of Prolog programs, makes it more attractive as a candidate for implementation at the microcode level or in hardware than the code required to support the WAM.

Both of these present possibilities for future investigation.

4.6 Acknowledgements

The authors also wish to acknowledge the support for this program of research provided (in many tangible and intangible ways) by Bill Hopkins, Lynette Hirschman, and the staff of the Logic-based Systems Branch of the Unisys Paoli Research center.

References

- [1] T. M. Blenko. *Compiling Logic Programs to Applicative Form*. Technical Memo 74, Paoli Research Center, Unisys Corporation, Paoli, PA, 1988.
- [2] J. P. Clark, D. P. McKay, and P. Duffy. *Prolog Benchmarks Report*. Technical Memo 35, Paoli Research Center, Unisys Corporation, Paoli, PA, 1986.
- [3] S. K. Debray. *The SB-Prolog System*. Tucson, AZ, December 1987. Version 2.2.1.
- [4] B. Fagin and T. Dobry. *The Berkeley PLM Instruction Set: An Instruction Set for Prolog*. Technical Report UCB/CSD 86/257, University of California, Berkeley, Berkeley, CA, September 1985.
- [5] E. Tick and D. Warren. *Towards a Pipelined Prolog Processor*. Technical Report, SRI International, Menlo Park, CA, August 1983.
- [6] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31-49, 1979.
- [7] D. H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

Section 5

**Or-Parallel Speed-Up
in Natural Language Processing:
A Case Study**

by

**Lynette Hirschman
William C. Hopkins
Robert C. Smith**

Table of Contents

Section	Title	Page
5.1	Abstract	5-1
5.2	Introduction	5-1
5.3	Background	5-2
5.4	Description of the Application	5-3
5.5	Simulation Models	5-4
5.5.1	Process State	5-6
5.6	Analysis of Results	5-7
5.6.1	Concurrency and Granularity	5-8
5.6.2	Speed-up	5-10
5.6.3	Effect of Overhead	5-12
5.6.4	Form of the Concurrency	5-12
5.6.5	Effect of Sentence Length	5-13
5.6.6	Linguistically Guided Concurrency	5-14
5.7	Conclusions	5-14
5.8	References	5-14

Table of Figures

Figure	Title	Page
5-1	Partial Parse Tree	5-5
5-2	Typical Unconstrained Concurrency	5-8
5-3	Typical Constrained Concurrency	5-9
5-4	Speed-up Curves	5-10
5-5	Process Length Distribution	5-11
5-6	Linear Parse Time	5-13

Table of Tables

Table	Title	Page
5-1	Example of a Simple Restriction Grammar	5-4
5-2	Process State	5-6
5-3	CASREPS Sentences used	5-7
5-4	Summary of Speed-Up Results	5-12

Section 5

Or-Parallel Speed-Up in Natural Language Processing: A Case Study¹

5.1. Abstract

We report on a series of simulation experiments for a large-scale natural language processing system. The results indicate that an or-parallel, all-solutions search provides substantial speed-up (20-30 fold) for this application. Longer sentences also show greater speed-up, giving parse times that increase *linearly* with sentence length. These results were obtained using a simple, application-specific model of independent, non-communicating or-processes in a shared memory environment. Simulations were run with a range of overhead costs; they show that significant benefits are obtained from parallel processing with overheads ranging as high as the median process size, although our estimates of overhead times are substantially smaller than median process size.

5.2. Introduction

We report here on a series of simulation experiments for a large-scale natural language processing system; our goal was to determine whether processing times in a large-scale application could be reduced substantially by selective exploitation of or-parallelism in a shared-memory environment. We were particularly interested in the granularity of process duration relative to overhead and in the distribution of opportunities for parallelism. We chose a simple, application-specific model of an all-solutions or-parallel search implemented as independent, non-communicating processes in a shared memory environment.

• The simulation results show substantial speed-up for this application: at 50% processor effectiveness (speed-up factor divided by number of processors), the speed-ups are typically in the 20-30 fold range. The longer the sentence, the greater the speed-up, giving parallel parse times that increase *linearly* with sentence length. Simulations were run with a range of overhead costs; they show that significant benefits are obtained from parallel processing with overheads ranging as high as the median process duration, although our estimates of overhead times are substantially smaller than median process duration.

We chose our application (a top-down parser with a broad-coverage English grammar) because it represented a large-scale logic program written in Prolog, developed independently of any notion of parallel processing. We limited our initial experiments to one particular type of application level or-parallelism: parallel exploration of

1. This section has been published in slightly different form in *Logic Programming: Proceedings of the*

disjunction (alternatives) in grammar rules, although there are many other opportunities for parallelism in the application.

5.3. Background

There have been a number of approaches for the implementation of or-parallelism, including the "Gigalips" project collaboration at Argonne Labs, the University of Manchester and the Swedish Institute of Computer Science [1, 2, 3, 4, 5], work at various institutions in Japan [6, 7], and research at a number of other sites [8, 9]. Or-parallelism has also been addressed within the model of Concurrent Prolog [10]. The goal of these researchers has been to support a transparent port of an application to a parallel Prolog (possibly with minimal annotation to indicate which goals should be run in parallel).

The goals of our research have been focused on finding a useful granularity of parallel in large-scale applications, rather than on the underlying or-parallel implementation. We therefore adopted a simple model of independent, non-communicating or-parallel processes, to minimize issues such as shared variable bindings. This model resembles the PIE model [6], and has some similarities with Shapiro's all-solutions approach in Flat Concurrent Prolog². The limited model of parallelism outlined here would also be appropriate for application of the Kabu-Wake method [7]. However, the grain-size differs from all of these approaches (except the Argonne model), since we focus on an application-level disjunction for spawning or-processes, rather than the finer-grained Prolog disjunction. Our work also differs from many of these [7, 5], in that process creation is determined solely by the annotations in the program, not by any run-time considerations. Dynamic splitting of processes on demand requires communication and structural overheads that are not required in our simpler model. The compensating cost is the overhead of handling more processes than may be necessary to keep the system busy.

One aspect of our application deserves mention, namely parsing to all solutions. Support for the all-solution approach to parsing becomes particularly important when applying broad-coverage grammars to speech understanding, where there is substantial indeterminacy in the input signal, necessitating massive exploration of parallel paths during parsing. The importance of the speech processing application alone would justify special architectures tailored for parallel parsing, although clearly use of an application-independent model of parallelism would be both more general and more flexible.

Our all-solutions application explains why our results show substantial parallelism, even with an overhead cost comparable in size to average process duration; it may also explain why our results differ from the negative results reported by Fagin

Fifth International Conference and Symposium, MIT Press, 1988.

2. In an all-solutions environment, or-parallelism merges with and-parallelism, because exploration of alternatives becomes mandatory: alternatives become indistinguishable from independent, non-communicating and-processes.

and Despain [11] who looked only at single solution search in most cases, and found, not surprisingly, little speed-up due to or-parallelism.

5.4. Description of the Application

Our experiments are based on the parser and grammar of the Unisys PUNDIT³ natural language processing system (Restriction Grammar) [12, 13]. The grammar consists of context-free BNF definitions, augmented with constraints, in the style of a logic-grammar. However, unlike Definite Clause Grammar [14] or its variants, there are no explicit parameters in Restriction Grammar. Contextual information is stated as well-formedness constraints on the parse tree, which can be freely traversed by a special set of "restriction operators" which examine the shape of the parse tree. To support the free tree traversal, the parser maintains a complex data structure consisting of both the parse tree and a path up to the root of the tree (for traversal up the tree), making grammar rule execution more complex (and larger grained) than a comparable operation in a plain DCG. The Restriction Grammar parser is implemented by an interpreter (instrumented for debugging) and a dynamic translator (for parsing with a stable grammar) [15].

We used the Restriction Grammar interpreter as the basis for the simulation experiments, because it supported the instrumentation required to generate input to the simulator. In order to obtain timing data to support the simulations, we modified the grammar interpreter slightly to use a continuation-based interpreter. At each grammar rule disjunction, the interpreter constructs an explicit continuation to be passed to the new process. Thus each disjunction becomes a self-contained process which runs to completion (or failure) completely independently of other processes. All that is required for a process spawn is a "clean copy" of the continuation data structure, with logical variables in this structure named apart from logical variables in other processes. This removed any dependency on the Prolog stack and enabled us to run crude timing experiments on the copying operation.

The English grammar consists of approximately 125 BNF definitions and 55 restrictions. This base grammar is augmented by a meta-rule which generates additional rules for handling conjunction, increasing the size of the grammar by about 50%. The coverage of the grammar is very broad; it includes an extensive treatment of coordinate conjunction [16], detailed coverage of complex subject and complement types, fine-grained analysis of noun phrases, treatment of relative clauses and questions, and also a small set of definitions for handling fragmentary sentences (e.g., *Metal particles in oil* or *Suspect coupling to be sheared*).

The result of the broad-coverage of the grammar is that parsing involves a good deal of search. There are approximately 350 disjunctions in the grammar (a branching factor of about 3). Many sentences receive multiple parses; even sentences receiving only a single parse still explore many alternative paths that end in failure.

3. *Prolog Understands Integrated Text.*

The Restriction Grammar interpreter consists of a clause defining the parser actions for each type of connective that can be found in a grammar rule: conjunction, disjunction, terminal nodes (indicated by an asterisk), literals, and restrictions (indicated by curly braces). Table 5-1 shows a highly simplified grammar, illustrating rules with conjunction (*sentence*, *noun_phrase*, *left_mods*, etc.), and disjunction (e.g. *right_mods*, *article*, *object*). The grammar disjuncts are the level at which we search for all solutions.⁴ Figure 5-1 shows the corresponding parse tree for a short sentence.

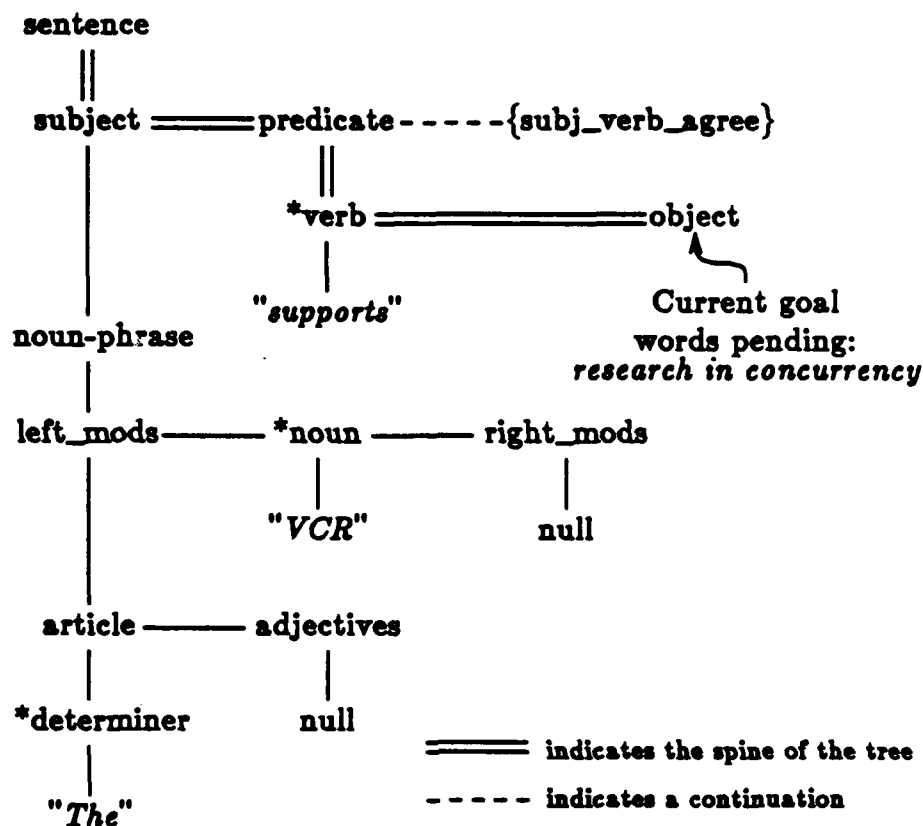
5.5. Simulation Models

The experimental approach uses a model-based simulation system named the *Virtual Computation Recorder* or *VCR* [17], which infers the concurrent behavior of the parser from its sequential behavior. Two processing models for the PUNDIT parser drive the simulation effort. The *sequential execution model* is derived from sequential Prolog execution, and defines the granularity of the simulations. The *concurrent execution model* is a special case of a shared-memory OR-parallel execution model for Prolog. These models and their relationship are a key element of the approach.

Sample Grammar		
<i>sentence</i>	::=	<i>subject</i> , <i>predicate</i> , { <i>subj_verb_agree</i> }.
<i>subject</i>	::=	<i>noun_phrase</i> .
<i>noun_phrase</i>	::=	<i>left_mods</i> , * <i>noun</i> , <i>right_mods</i> .
<i>left_mods</i>	::=	<i>article</i> , <i>adjectives</i> .
<i>right_mods</i>	::=	<i>prepositional_phrase</i> ; <i>null</i> .
<i>article</i>	::=	* <i>determiner</i> ; <i>null</i> .
<i>adjectives</i>	::=	* <i>adjective</i> , <i>adjectives</i> .
<i>adjectives</i>	::=	<i>null</i> .
<i>predicate</i>	::=	* <i>verb</i> , <i>object</i> .
	::=	* <i>verb</i> , { <i>be_verb</i> }, <i>be_predicate</i> .
<i>object</i>	::=	({ <i>transitive_verb</i> }, <i>noun_phrase</i>); ({ <i>intransitive_verb</i> }, <i>null</i>).
<i>be_predicate</i>	::=	* <i>adjective</i> ; <i>prepositional_phrase</i> .
<i>prepositional_phrase</i>	::=	* <i>preposition</i> , <i>noun_phrase</i> .
<i>null</i>	::=	~ . % Marks empty element.

Table 5-1: Example of a Simple Restriction Grammar.

4. There are many additional opportunities for or-parallelism, as well as for independent and-parallelism that we have not exploited in this simulation; this will be the subject of future work.



The VCR supports research in concurrency

Figure 5-1: Partial Parse Tree.

The PUNDIT parser searches for all parses of the input string, using standard Prolog backtracking to explore the search tree completely. A single parse tree is built and unbuilt in backtracking, so there is only one partial parse tree, representing the state of the currently active parse, at any time.

The concurrent execution model of the parser supports the concurrent construction of independent parse trees, spawning processes in shared memory for the disjunctive goals encountered. Each child process will attempt to complete the partial parse supplied by its parent using one of the goals. Spawning is under the control of a policy which specifies which disjuncts are spawned and which are left to be pursued with the normal failure/backtrack mechanisms. At one policy extreme (never spawning), the model is that of sequential Prolog; at the other (always spawning), no backtracking occurs, as each disjunct is explored by its own process. Failure to satisfy a spawned goal results in termination of the process. The cost of backtracking and of

maintaining backtracking information can thus be avoided for spawned goals. For simplicity, the rest of the discussion will discuss only spawned disjuncts. Any unspawned disjuncts are treated with the normal Prolog mechanisms.

The models are not concerned with the details of satisfying a goal beyond identifying subgoals and spawning processes for them. Thus, only four types of processing are identified in the sequential execution of the PUNDIT parser: *start-goal*, which sets a goal, *complete-goal*, the successful derivation of the goal, *fail-goal*, the failure to derive the goal, and *backtrack*, the search back to the most recent alternative goal. Annotations in the PUNDIT source identify the transitions from one type of processing to the next; these annotations serve two purposes: to generate the behavior description in terms of chunks of processing, which are called *atoms*, and, in a separate step, to allow timing of the atoms.

Each disjunct represents a distinct attempted parse, and has its own parse tree, which is initially a copy of the parse tree of its parent. The result of a concurrent parse is the forest of parse trees produced by the successful parses.

5.5.1. Process State A child process receives the parse tree computed so far, the continuation stack (specifying the remaining conjoined subgoals for the goal being sought at each level of the tree), the return path to the root of the parse tree, the current goal, and the remaining input string. No other data from the parent process is needed. The concurrent execution model exploits sharing between parent and child of the parts of this data which contain no unbound logical variables when the child is spawned; only those portions are copied that may contain unbound logic variables, as outlined in Table 5-2.

As the parse proceeds, the tree is expanded along its right edge. All unbound logic variables are on the spine (the path from the root to the current site of tree-building), so the completed subtrees to the left of the spine are constant, and may be shared. The return path is the inverse of the spine, containing back pointers along the path to the root. The parse tree (Figure 5-1) for the simplified grammar example identifies the spine and return path with doubled links. The spine itself must be copied, renaming apart the logical variables in it for the child process; a new return path corresponds to the new spine. The continuation stack contains no logical variables and may be shared.

State component	Shared	Local
parse tree	subtrees left of spine	spine
return path	-	return path
continuation stack	continuation stack	head pointer
remaining words	wordstream	"next" pointer

Table 5-2: Process State.

To determine the cost of performing the necessary copying and naming apart of variables, the spine of the tree and the return path were copied explicitly in Prolog. The spine grows and shrinks during the course of parsing and exact copying time depends on the actual length of the spine when copied. We measured the copying times during an actual parse; the average copying time was measured to be about 1 millisecond. This corresponds reasonably well with intuition (approximately 50-100 nodes copied, with one logical inference required per node copied on a 50K Lips Prolog). The copying time increased only slightly with the length of the spine, indicating that other costs were being included; we have, however, taken the conservative path of using the measured copying times. The time to start up the spawned process is arbitrarily set at 1 millisecond to allow for process creation and system overheads. All these timings assume a simple model of parallel Prolog in which the specialized copying is not supported as a primitive; directly implementing such a primitive could be expected to reduce the copying time substantially.

5.6. Analysis of Results

We performed a large number of experiments, varying the sentence being parsed, the number of processors available, the times for processing atoms, and most important, the overhead associated with spawning a new process. The sentences are listed in Table 5-3; this discussion will use the results for a typical sentence, *Unit has excessive wear on inlet impeller assembly and shows high usage of oil*. Full results, with extensive data, are available in [18]. All results here assume a process spawn on every grammar rule disjunction.

Test Sentences	
1.1.1	starting air regulating valve failed.
4.1.1	while diesel was operating with sac disengaged, the sac lo alarm sounded.
4.1.3	pump will not turn when engine jacks over.
5.1.2	disengaged immediately after alarm.
6.1.2	inspection of lo filter revealed metal particles.
9.1.1	sac received high usage during two becce periods.
9.1.4	loud noises were coming from the drive end during coast down.
22.1.1	loss of lube oil pressure during operation.
25.1.3	suspect faulty high speed rotating assembly.
28.1.1	unit has excessive wear on inlet impellor assembly and shows high usage of oil.
31.1.3	erosion of impellor blade tip is evident.
31.1.4	compressor wheel inducer leading edge broken.

Table 5-3: CASREPS Sentences used.

CONCURRENCY (unlimited processors)

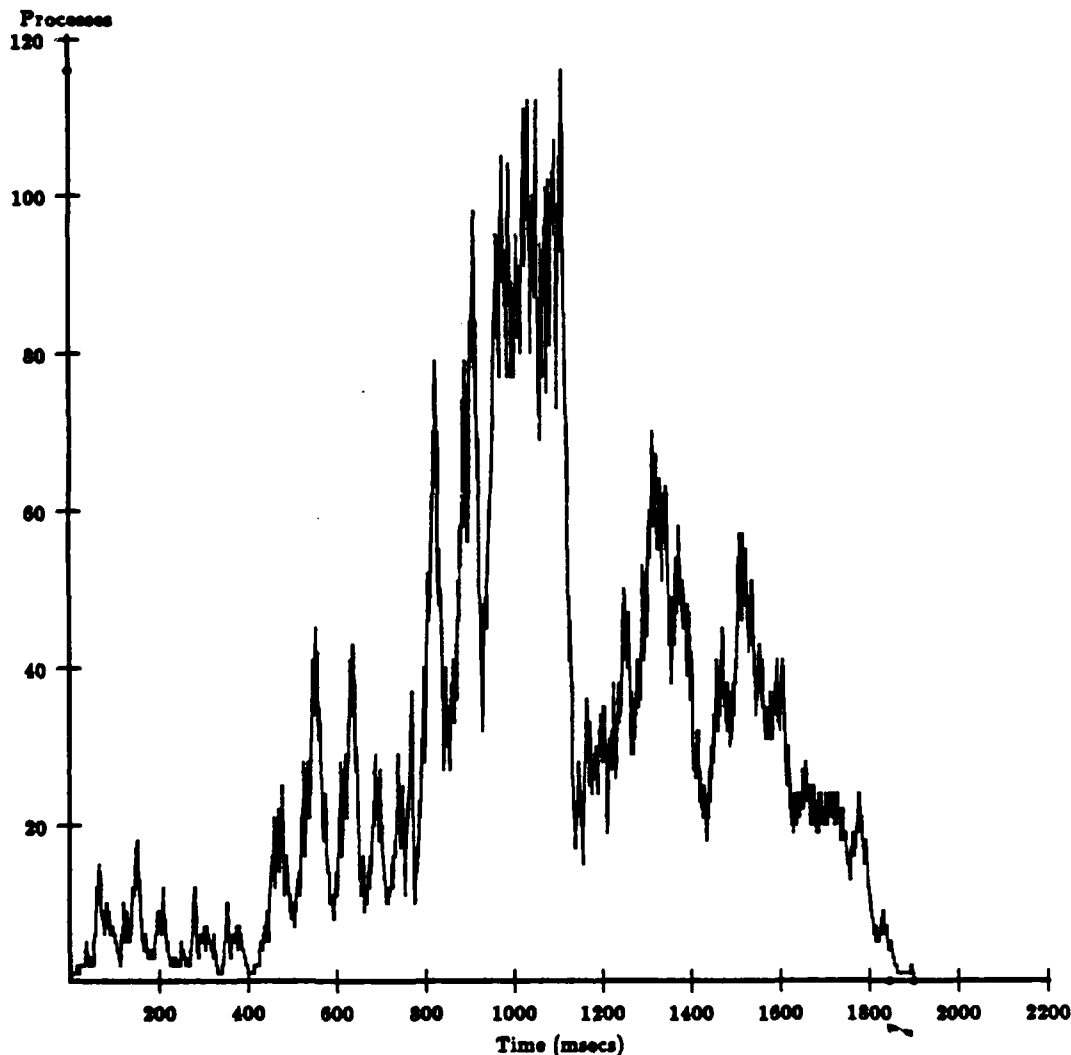


Figure 5-2: Typical Unconstrained Concurrency

5.6.1. Concurrency and Granularity Figure 5-2 shows the concurrency discovered in the parsing of a typical sentence; it graphs the number of active processes over time, assuming enough processors (116) so that one is available to run each process when it is spawned. As is evident in the example, the concurrency is irregular, with one or more peaks and potentially long start-up and shut-down tails. Furthermore, the shape of the concurrency varies markedly from sentence to sentence, depending on where in the sentence multiple parses are attempted and how long the parser follows a path before the path ends in failure.

Figure 5-3 shows the active process distribution for the same sentence with a 50 processor limit, which increases the parse time by about 10% from the unlimited case. Processes that are spawned when all processors are busy wait in a strict (FIFO) queue. The effect of varying the number of processors on speed-up is shown in Figure 5-4; the curves have a characteristic shape with a distinct knee that reflects the decreasing effectiveness of additional processors. Below the knee, speed-up is close to linear; beyond it, the waiting queue empties and additional processors are increasingly idle.

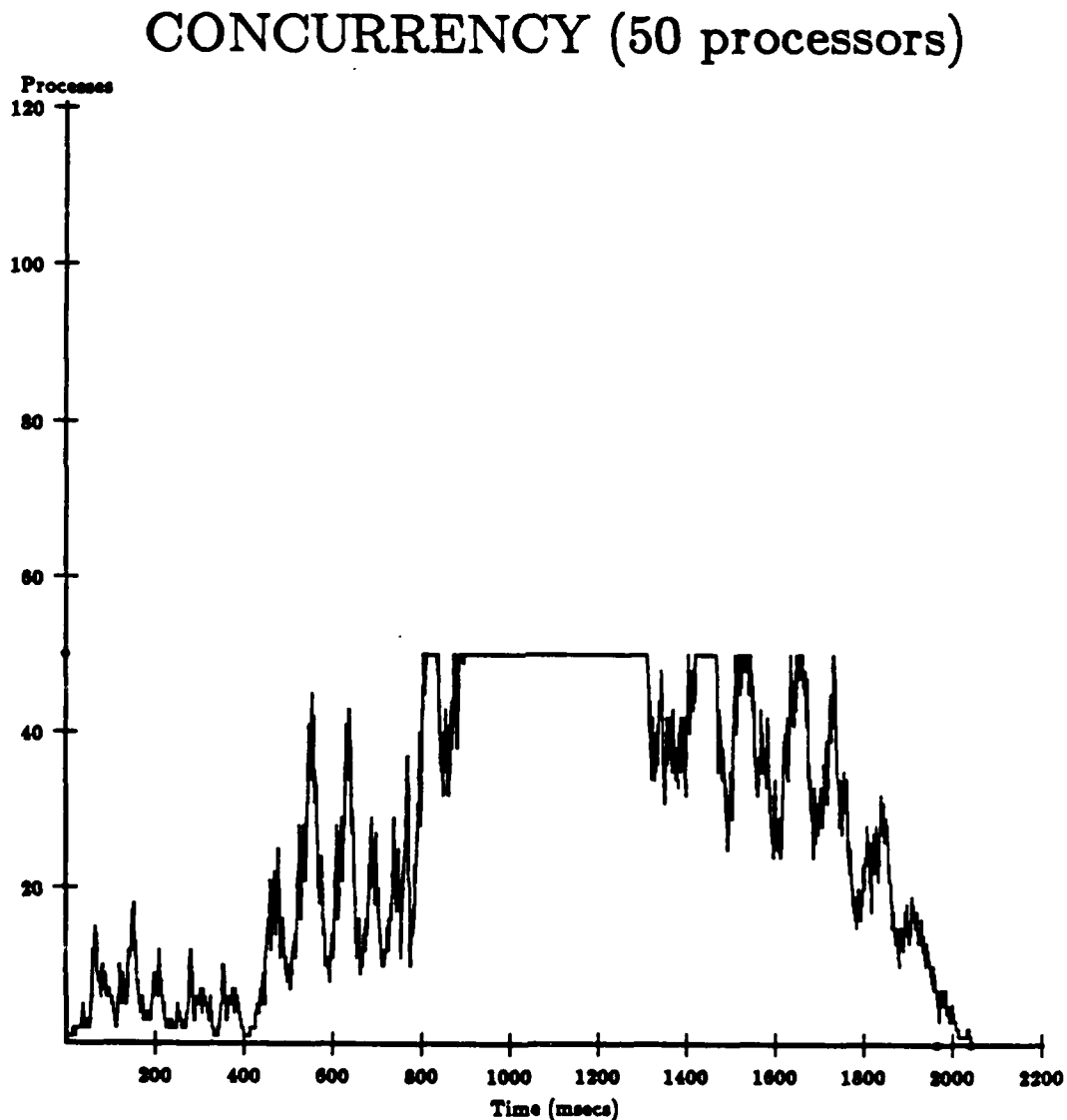


Figure 5-3: Typical Constrained Concurrency

Speedup (1ms startup, variable copying)

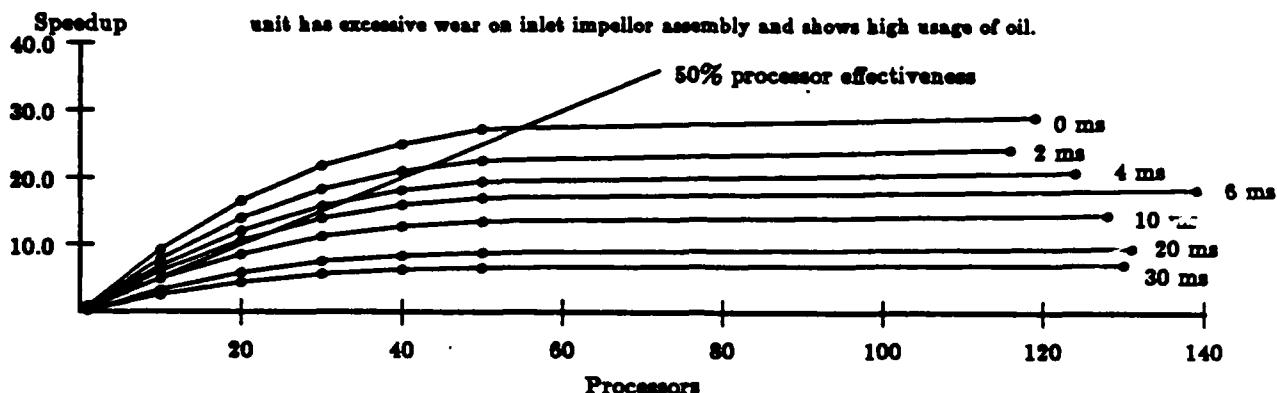


Figure 5-4: Speed-up Curves

The duration of the processes and the relative cost of spawning them are extremely significant for matching the processing to a multiprocessor architecture. Our results here are quite encouraging; the median process time, using measured atom timings from interpreted Prolog runs, is typically around 9 milliseconds. Figure 5-5 shows the distribution of process duration in 5 millisecond increments; it is remarkably invariant over the sentences tested.

5.6.2. Speed-up The calculation of parallel speed-up factors, especially the selection of the base time, is notoriously open to "optimization of results." The generally accepted rule is to compare to the best sequential implementation, to get a fair measure of system design choices. We take the sequential parse times of the unmodified PUNDIT parser as our standard. (The instrumentation technique required using a grammar interpreter written in PROLOG; an additional doubling of speed is obtainable by translating the grammar directly to PROLOG. Section 6 addresses the implications of this speed-up.) The timings are reconstructed from the measured behavior and atom timings of the parser (as modified for continuation stacking) and to normalize for the systematic effect of the measurements. The effect of the parser modifications and the instrumentation is to increase the execution time by about 50%; to compensate, we increase the copying overhead (by more than 50%) to 2 milliseconds to maintain the proper (conservative) ratio. All timings are derived from Quintus Prolog 1.5 running on a Sun 3/160 with 16 megabytes of memory.

Figure 5-4 shows a family of speed-up curves for the example sentence. The speed-up factor over the sequential execution time is shown as a function of the number of processors available. The additional parameter, which differs among the several curves, is the copying overhead, discussed in the next paragraphs. The straight line identifies 50% processor effectiveness, the points at which the speed-up

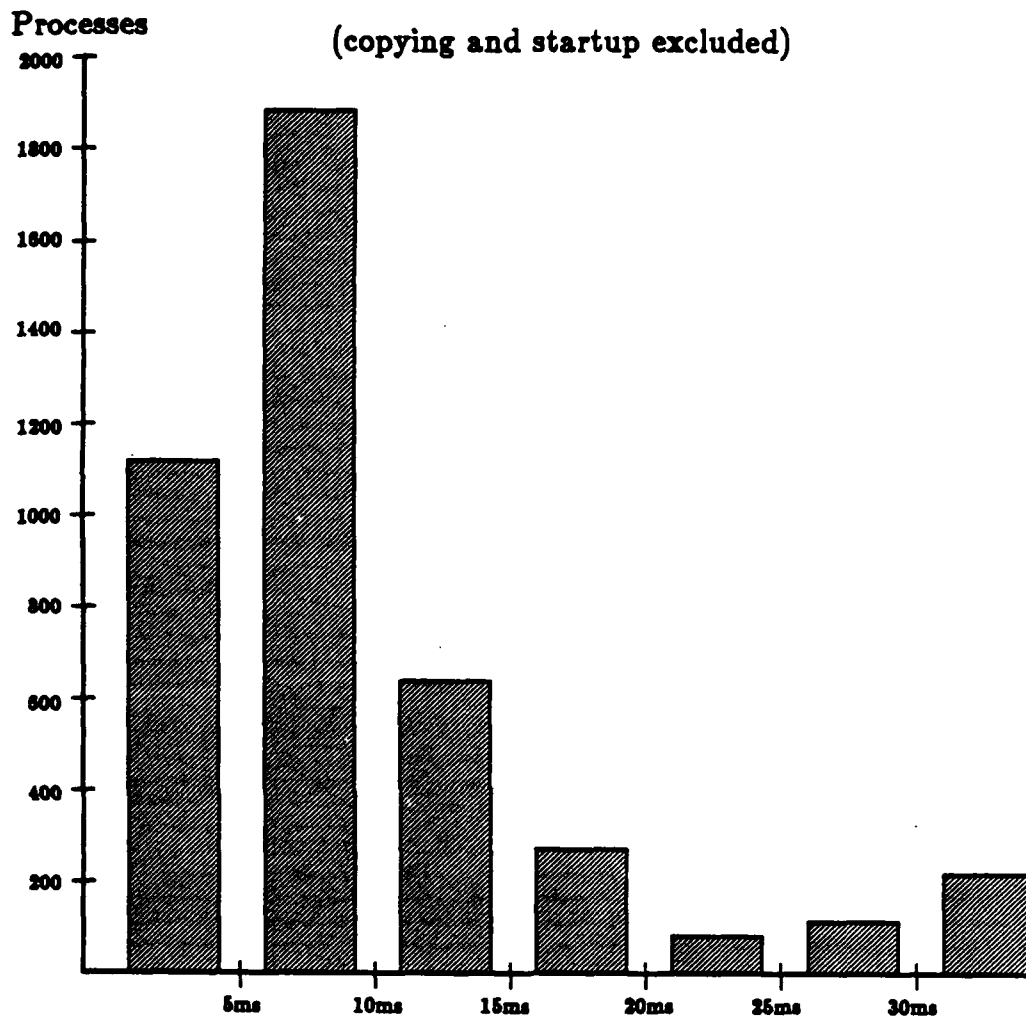


Figure 5-5: Process Length Distribution

factor is half the number of processors⁵. This arbitrary limit is taken as the threshold for acceptable system performance; it typically is at or near the knee of the speed-up curve. Table 5-4 summarizes the speed-up for the sentences parsed.

5. Processor utilisation, which merely measures how busy the processors are, makes an apparent virtue of overhead. Processor effectiveness, sometimes called *efficiency*, is the average *useful* processor utilisation.

PUNDIT Concurrency Results (2ms copying 1ms startup)							
Sent.	lth.	time(sec.)		maximum		50% effect.	
		seq.	par.	spd.	proc.	spd.	proc.
5.1.2	4	10.8	0.9	11.8	62	-	-
1.1.1	5	7.3	0.7	10.1	42	9.9	20
25.1.3	6	26.8	1.5	18.5	121	-	-
31.1.4	6	24.9	1.1	23.2	71	-	-
31.1.3	7	6.1	0.9	7.1	28	6.9	14
6.1.2	7	7.8	0.9	8.5	37	7.9	16
22.1.1	7	22.5	1.9	11.6	59	-	-
4.1.3	8	7.4	1.2	6.4	37	5.8	12
9.1.1	8	13.7	1.2	11.9	42	11.6	23
9.1.4	11	37.3	1.4	26.1	93	22.6	45
4.1.1	12	51.5	1.6	31.8	153	27.8	56
28.1.1	14	46.2	1.9	24.3	116	21.1	42

Table 5-4: Summary of Speed-Up Results.

5.6.3. Effect of Overhead Figure 5-4 shows the speed-up curves for our example sentence under five different copying overheads, ranging from 0 to 30 milliseconds; the process start-up is constant at 1 millisecond. As can be seen, if the copying time exceeds 4 milliseconds, more than a third of the potential speed-up is lost; at 10 milliseconds, more than half is gone. This suggests that parallelism can be effectively exploited if process overhead is kept to a fraction of the median process duration (here, 9 milliseconds).

Shen and Warren report simulation results [3] that show the same characteristic curves (their Figure 5-3) using an artificial time unit of a "resolution time", with the cost of process spawning usually being four resolutions (some tests vary it from 0 to 16 resolutions). They also find that the overhead has a substantial effect on the achievable speed-up. This leads us to hypothesize that the key parameter relating overhead to speed-up is the ratio of process initialization overhead to median process duration; this remains to be confirmed by additional experimentation.

5.6.4. Form of the Concurrency Spawning separate processes for grammatical disjuncts usually has the effect of starting the exploration of the "correct" parse sooner than would happen sequentially (unless it is the left-most; the grammar has indeed been written to encourage this behavior, which is desirable in a sequential system). Spawning all disjuncts has the effect that for any potential parse, there is a series of processes that attempts that parse with no unnecessary searching. Given sufficient resources, this will accomplish the ultimate goal of reducing the time to develop the

correct parse.

5.6.5. Effect of Sentence Length Figure 5-6 shows the relationship between sentence length and parallel parsing time with unlimited processors. For these thirteen sentences, the relationship appears to be linear, which is consistent with the processing model. Since each potential parse is performed as quickly as possible, the time to complete it is proportional to the number of nodes in the parse tree; this is very close to the number of words (being at worst $N \log N$ for N words).

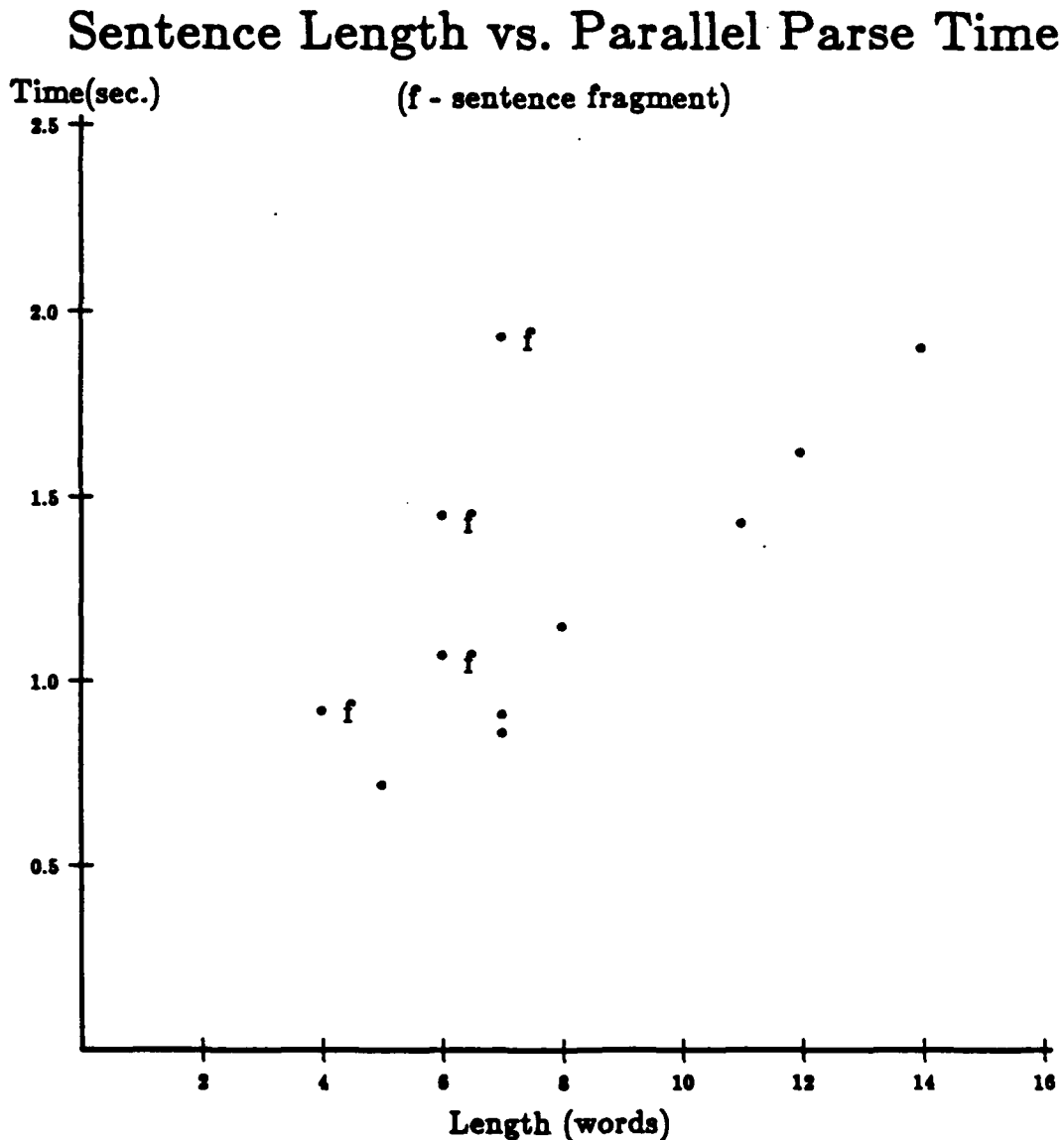


Figure 5-6: Linear Parse Time

5.6.6. Linguistically Guided Concurrency Our original intent was to investigate the use of static linguistic knowledge to guide spawning, e.g., spawn only on certain "rich" nodes or nodes with a high probability of success. However, the results from the simpler always-spawn policy were sufficiently good that we have deferred investigation of other policies. As process duration is reduced, however, we may return to this line of investigation as a way of increasing process granularity relative to overhead.

5.7. Conclusions

We have demonstrated exploitable or-parallelism in an important real Prolog application, and have simulated significant speed-up with an application-specific parallel execution model. For a small but representative set of sentences, we have observed linear-time parsing in the unlimited-processor case.

Some necessary work remains to sharpen and improve the simulation model and parameters. We have previously mentioned the instrumentation overhead that remains in the atom duration measurements. We have also not treated garbage collection properly, as we have not yet dug into the Prolog implementation to identify and measure it precisely to allow prediction of a multiprocessor equivalent. The overhead of maintaining choice points and trail information, unnecessary for spawned disjuncts, is another unknown. These effects may reduce the granularity of the processes, making the overhead more significant if it is not further optimised. Finally, the measurements are made on a parser that interprets the grammar, rather than running the more efficient version of the grammar obtained by translating it to PROLOG. This typically doubles the parser speed, so speed-up projections for this case will be consistent with the results reported here for the 4 millisecond copying cost.

The results described here apply to an application-specific large-grained form of concurrency. Research in related areas [19, 3] points to the existence of finer-grain concurrency within the general or-parallel model that would apply to the computations that we have taken as atomic. Further research is necessary to determine the amount of additional speed-up that can be gained by combining approaches.

We have recently begun collaboration with the researchers at the Swedish Institute of Computer Science; our plan is to furnish the application described here, for benchmarking on the Aurora Or-parallel machine. This work should provide us with the ability to validate our simulation results. It should also provide insight into speed-ups obtainable on large-scale applications in a general or-parallel implementation.

5.8. References

- 1 A. Ciepielewski, S. Haridi, and B. Hausman, Initial Evaluation of a Virtual Machine for OR-Parallel Execution of Logic Programs. *Proceedings of IFIP TC-10, U. of Manchester*, 1985.

- 2 Terry Diss, Ewing Lusk, and Ross Overbeek, Experiments with Or-Parallel Logic Programs. In *Proc. of the Third International Conference on Logic Programming*, J. L. Lasses (ed.), Melbourne, 1987, pp. 576-600.
- 3 K. Shen and D. H. D. Warren, A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proc. of the Third International Conference on Logic Programming*, J.-L. Lasses (ed.), Melbourne, Australia, 1987, pp. 54-68.
- 4 D. H. D. Warren, The SRI Model for Or-Parallel Execution of Prolog - Abstract Design and Implementation. In *Proc. of the 1987 Symposium on Logic Programming*, The Computer Society of the IEEE, San Francisco, CA, 1987, pp. 92-102.
- 5 B. Hausman, A. Ciepielewski, and S. Haridi, OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *Proc. of the 1987 Symposium on Logic Programming*, San Francisco, 1987, pp. 69-79.
- 6 T. Moto-oka, H. Tanaka, H. Aida, and T. Maruyama, The Architecture of a Parallel Inference Engine - PIE. *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984, pp. 479-488.
- 7 Y. Sohma, K. Satoh, K. Kumon, H. Masusawa, and A. Itashiki, A New Parallel Inference Mechanism Based on Sequential Processing, TM-0131, ICOT, 1985.
- 8 J. S. Conery, Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. In *Proc. of the 1987 Symposium on Logic Programming*, San Francisco, 1987, pp. 457-467.
- 9 H. Westphal and P. Robert, The PEPSys Model: Combining Backtracking, AND- and OR-Parallelism. In *Proc. of the 1987 Symposium on Logic Programming*, San Francisco, 1987, pp. 436-448.
- 10 Ehud Shapiro, An Or-Parallel Execution Algorithm for Prolog and its FCP Implementation. In *Proc. of the Third International Conference on Logic Programming*, J.-L. Lasses (ed.), Melbourne, Australia, 1987, pp. 311-337.
- 11 B. S. Fagin and A. M. Despain, Performance Studies of a Parallel PROLOG Architecture. In *Proceedings of the 14th International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June, 1987, pp. 108-116.
- 12 L. Hirschman and K. Puder, Restriction Grammar: A Prolog Implementation. In *Logic Programming and its Applications*, D.H.D. Warren and M. VanCaneghem (ed.), 1985, pp. 244-261.
- 13 Martha S. Palmer, Deborah A. Dahl, Rebecca J. Schiffman, Lynette Hirschman, Marcia Linebarger, and John Dowding, Recovering Implicit Information. In *Proc. of the 24th Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, August 1986.
- 14 F. C. N. Pereira and D. H. D. Warren, Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13, 1980, pp. 231-278.
- 15 John Dowding and Lynette Hirschman, Dynamic Translation for Rule Pruning in Restriction Grammar. In *Proc. of the Second International Conference on Natural Language Understanding and Logic Programming*, Vancouver, August, 1987.

- 16 L. Hirschman, Conjunction in Meta-Restriction Grammar. *J. of Logic Programming* 4, 1986, pp. 299-328.
- 17 W. Hopkins and R. Smith, Concurrency Simulation by Abstract Interpretation, LBS Technical Memo, Paoli Research Center, Unisys Corp., Paoli, PA, December, 1987.
- 18 Lynette Hirschman, William C. Hopkins, and Robert C. Smith, Simulation of Or-Concurrency in the PUNDIT Parser, LBS Technical Memo, Paoli Research Center, System Development Corp., Paoli, PA, June, 1988.
- 19 T. Blenko, Compiling Logic Programs to Applicative Form, LBS Technical Memo 74, Paoli Research Center, Unisys Corporation, Paoli, PA, January, 1988.

Section 6

**Concurrency Simulation
by Abstract Interpretation**

by

**William C. Hopkins
Robert C. Smith**

Table of Contents

Section	Title	Page
6.1	Introduction	6-1
6.1.1	Problem Statement	6-1
6.1.2	Objective	6-1
6.1.3	The Virtual Computation Recorder	6-2
6.1.4	Example of VCR Use	6-2
6.2	Related Work	6-4
6.3	VCR Concepts	6-4
6.3.1	The Behavior Model	6-4
6.3.2	Behavior Description and its Representation	6-5
6.3.3	Abstract Interpretation	6-5
6.3.4	Reinterpretations	6-5
6.4	VCR Implementation	6-6
6.4.1	Recording and Graph Building	6-6
6.4.2	Playback	6-8
6.4.3	Reinterpretations	6-9
6.4.4	Analysis	6-10
6.5	Conclusion	6-10
6.5.1	Benefits of the approach	6-12
6.5.2	Limits of the approach	6-12
6.5.3	Directions for future work	6-13
6.6	References	6-14

Table of Figures

Figure	Title	Page
6-1	VCR Structure and Flow	6-7
6-2	Typical Process Length Distribution	6-11
6-3	Typical Constrained Concurrency Graph	6-11
6-4	Typical Process Length Profile	6-11
6-5	Typical Speed-up Graph	6-11

Table of Tables

Table	Title	Page
6-1	Atom Behavior Classes	6-8

Section 6

Concurrency Simulation by Abstract Interpretation

6.1. Introduction

In the study of parallel system architectures, many degrees of freedom are available to the researcher. Experiments, however, are often extremely difficult, time-consuming, and expensive to set up, and provide only a few data points in a large and complex solution space. We report a tool that is currently in use to aid the search for empirical data about a broad, but not complete, class of parallel system architectures, both hardware and software, with a primary emphasis on identifying concurrency exploitable on a parallel architecture. The system helps define the architectural requirements by facilitating the trade-off and "what-if" analysis of the system architecture design.

6.1.1. Problem Statement

Our model of research is based on the concept of a *computation*, which we will use rather broadly, and initially not very strictly; later on we will restrict it as necessary. Intuitively, a computation is what a computer does in executing a program. The computation is assumed to be decomposable into constituents with measurable duration and a partial ordering in time which may be based on data dependencies, control flow, or other considerations. To study an application area, an existing computation that currently runs sequentially, or a family of such computations, is obtained by running programs from the area. The problem is to identify portions of the computation that may run in parallel, and to map them onto a multiprocessor system to gain an effective speed up.

A wide variety of concurrent execution models and concurrency policies are available to the researcher; the choice among them should be based on reliable information about their effectiveness. A major obstacle to success is the inability to obtain experimental evidence of the effectiveness of any particular approach without a substantial investment either in an implementation or in a model-specific simulation system. This typically requires an early commitment to the first model or policy chosen, a commitment that is often difficult to change later; this effectively restricts the breadth of study rather severely.

6.1.2. Objective

The tools described here have been developed in response to this problem. The specific objective is to provide rapid implementation of a simulation system that makes only minimal commitments to any specific choice of model or policy. One key to achieving this objective is the early identification of the degrees of freedom that are to be explored, and those that are fixed; the simulation system, to be effective, must be non-committal on the former. Subsidiary objectives include ease of use, ability to

specify experimental measurements, and efficiency of operation.

6.1.3. The Virtual Computation Recorder

Our approach to solving the simulation problem is called the *Virtual Computation Recorder*, or *VCR*. Overloading the abbreviation is intentional, with video-cassette technology as a metaphor for the simulation system; we believe that it is helpful, if limited. The VCR addresses the problem by allowing the simulation of concurrent implementations of applications without requiring that the concurrent system be implemented, or in some cases, fully designed. Instead, a model of the parallel system architecture is used to derive an application's behavior on the parallel system from the sequential behavior of its computation.

The VCR concept is built around a strong, user-specified model of the problem to be studied. The model identifies the constituents of the computation, their necessary ordering, and fixed aspects of the concurrency policies. We call the constituents *atoms*, to capture the idea of their indivisibility within the model. Based on the model, the existing computations are instrumented to identify the atoms, and the behavior of the computation is recorded in terms of the atoms. This recording, or *behavior description*, becomes the basis for simulation. Specification of the atoms involves a trade-off between flexibility and efficiency; they should be small enough not to hide potential concurrency, but not so small as to make the behavior description unmanageably large. It is also desirable that recurring atoms have constant duration, although this need not always be so.

The behavior description is not a representation of the program logic, but a record of its behavior for a particular set of inputs. All of the run-time tests and branching and the various other types of processing that the program may perform are undifferentiated and anonymous unless the atom set specifically identifies them. Interpretation of the behavior can therefore be made extremely simple and efficient. Only those aspects of the behavior that are relevant to the research questions at hand are addressed if the model and atom set are properly designed. It is obviously extremely important that these aspects be identified and thought through early in the research.

Instantiation of the unbound aspects of the model is accomplished through abstract interpretation of the behavior description, creating a new, more specific behavior description. Several varieties of abstract interpreters are possible, including *playback* devices, which instantiate the time of occurrence attribute of the atoms in the behavior description.

6.1.4. Example of VCR Use

In the following sections, we elaborate on the concepts introduced above. For intuition, we shall draw on the two instances of the VCR that we have implemented. These are briefly described in the following paragraphs.

Combinator-Graph Reduction

As part of the MASC program, the Paoli Research Center is investigating effective techniques for combinator-graph reduction. Combinator reduction is an implementation technique for lazy functional languages [1] [2] [3]

which can be supported in hardware [4]. Combinators are "pure functional forms," with bound variables abstracted out. They capture at a low level the concepts of lazy evaluation, referential transparency, and higher-order functions, and expose the concurrency that is derivable from the Church-Rosser property. A combinator graph consists of vertices and edges, where each vertex represents the application of a function (a combinator or a subgraph representing a combinator expression) to one or more arguments, all specified by directed edges from the vertex; leaf vertices represent values. (The necessary intuitions can be drawn from a simpler model where an interior vertex represents an operation and the edges designate its operands.) Sharing of the value of a common sub-expression is simply supported by directing several edges to the root of the sub-graph. Reduction of the combinator graph to standard form proceeds through a series of graph rewritings, eventually yielding the value of the program. The graph reducer used in this research is written in C.

The CR model for combinator-graph reduction uses a fine-grained set of atoms, to allow investigation of the effect of changes in the fine structure of the reduction process. The model of concurrent processing supports the necessary interaction of concurrent processes through the graph. Transformations of the recorded behavior have been implemented to identify opportunities for concurrent evaluation of multiple strict arguments of a combinator (addition, for example, which requires two evaluated arguments), and to collapse multiple atoms into larger-grained atoms, to reduce the size and complexity of the behavior description.

Natural Language Parsing

Unisys research in Natural Language Processing has resulted in the PUNDIT system for understanding English text in any of several problem domains. The PUNDIT parser [5] is a grammar-driven top-down parser, written in Prolog, that produces one or more parse trees for semantic analysis. An extended BNF grammar provides broad coverage of informal English text. Many input strings are syntactically ambiguous (e.g., "Racing cars can be dangerous"), so the parser generates all possible parse trees for discrimination during semantic processing. The concurrency model uses OR-parallelism, based on alternatives in the grammar rules (e.g., a sentence can be an assertion, a question, or a fragment). The concurrent PUNDIT parsing model pursues these alternatives with independent processes, each of which may complete a full parse. This allows a fairly coarse-grained set of atoms, and a simple model of concurrent processing.

6.2. Related Work

The idea of capturing and replaying the behavior of a program is not new. We are aware of several systems which do this for one application or another. Perhaps the most significant independent development is SPAN [6], an IBM system for analyzing parallel Fortran performance. It is very tightly tied to the VM/EPEX Fortran system; there does not appear to be any plan to generalize it. The compiler plants flashpoints in sequential code to identify the bounds of separable tasks, which are explicitly identified by the programmer. Execution of the flashpoints generates a task-level trace which is manipulated to expose the sequential and parallel constraints on the tasks; the problem is then treated as a multiprocessor scheduling problem. Simulation of the parallel behavior of the program is limited to the architecture of interest.

Another system of some note is "Instant Replay" [7], which use Behavioral Abstraction [8] in the debugging of parallel programs. The emphasis here is on the ability to debug multiprocess programs running on a multiprocessor by replaying suspicious behavior.

6.3. VCR Concepts

Intuitively, the VCR records program behavior and plays it back on a multiprocessor system, with control over system characteristics and parameters. Key concepts in the approach are the behavior model and its implications, and abstract interpretation.

6.3.1. The Behavior Model

The computational model defines the experimental world for the VCR. It defines the quantities to be directly measured and all parameters which may be varied, or which significantly affect the measured quantities. The base model, on which specific models are built, defines atoms and processes which interpret them. Atoms are units of sequential processing with a partial ordering such that the interpretation of an atom by a process leads to the interpretation of its successors. Atoms have attributes which are instantiated (bound) either initially or through interpretation. Initial attributes include the class and successor attributes. The class helps determine the order of interpretation, and includes *spawn*, procedure calls (*evaluate* and *return*), etc. (see section 3.2). Atoms generally have other attributes, such as type, duration, and various flags, that are defined by the specific model.

An initial set of atoms is defined in the sequential computation through annotations of the program text or other means. For the VCR instances described, the processing represented by atoms is delimited by annotations in the C and Prolog applications. The annotations are used both to generate the behavior description and to generate time duration statistics for the atoms.

6.3.2. Behavior Description and its Representation

Given a model and a computation, a behavior description is the set of atom instances that the computation entails and the ordering that the computation imposes. A sequential computation will generate a trace of the atoms encountered in the program execution, which imposes a complete ordering on them; a concurrent computation will generate concurrent traces, imposing a partial ordering.

Various representations of the behavior are possible, with varying amounts of information. A sequential trace, for instance, captures the full linear ordering of a sequential execution of a computation, but does not admit any concurrency unless explicitly salted with spawn atoms, which specify labeled entry points in the trace to be activated. The VCR also uses an alternative form, named an *Abstract Behavior Graph* or *ABG*, to represent a less restrictive partial ordering. An ABG is a directed graph with vertices representing atoms and edges designating successors; spawn atoms represent explicit concurrency in the behavior. Other representations are, of course, possible. The choice of representation depends on what information is of interest and the kind of processing to be performed on it. The VCR currently uses several forms of trace as well as the ABG, according to the problem and questions being addressed.

6.3.3. Abstract Interpretation

Abstract interpretation assigns meaning to atoms and computations. The atoms themselves represent pieces of the computation; their interpretation by the VCR is non-standard in the sense that it ignores the original intent of the computation, which is to produce the result of the program. Instead, the interpretation assigns attributes to the atoms which are of interest in the experimental program, such as time of occurrence, process number, processor allocation, and so on. Attributes are also thus attributed to the computation, such as time to completion, concurrency, and processor utilization.

This process of abstract interpretation, without concern to the details of the original computation, allows the experimenter using the VCR to focus on the issues of concern. Changes in, for instance, the details of the spawning mechanism can be modeled very simply, by altering the times attributed to the atoms for spawning and starting up new processes. To obtain similar results with the system being modeled would presumably require substantial design, implementation, and debugging effort. This advantage greatly increases the experimenter's productivity, and allows a wider range of options to be evaluated. In particular, a wide variety of "what-if" questions can be answered quickly, allowing experimentation to concentrate on the areas of highest payoff.

6.3.4. Reinterpretations

Most VCR interpretations of a behavior description produce another description. The new description may be in the same form and representation as the previous one, or it may be entirely different. For instance, the concurrent playback interpretation may,

at the user's option, generate only a summary description, or it may generate a detailed interleaved trace of the the atoms interpreted. The term *reinterpretation* (more correctly, *reinterpretable intpretation*) denotes the process of generating an interpretable behavior description. Reinterpretations include building the ABG, transforming it back to a sequential trace, transforming it to a modified set of atoms, and instantiating its attributes through simulation or otherwise.

6.4. VCR Implementation

In this section, we describe the specific implementation of the VCR that we are using, and some of the implications of the implementation design choices. Figure 6-1 shows the overall VCR structure and methodology flow.

6.4.1. Recording and Graph Building

Recordings for the VCR are made by instrumenting the application with annotations to produce a trace of atoms (the behavior description). The annotation must identify essential program behavior necessary for concurrency experimentation and preserve execution duration characteristics (to the extent needed) while ignoring unnecessary detail. The atoms thus defined must provide sufficient information to allow a broad range of concurrency policies, yet be coarse enough to yield fast simulations to allow extensive experimentation with policies and parameters.

In the combinator graph reduction domain, annotations inserted in the interpreter (the graph reducer) expand to make it an instrumented interpreter to produce the recordings. Where a system uses a fixed interpreter, instrumentation of the interpreter applies to all application programs, and is relatively easy to accomplish. In the PUNDIT parser, annotations are added to the interpreter for the grammar rules, which invokes the instrumentation when it encounters the annotations and supplies information on the grammar rule being used. For compiled or directly executable programs, other instrumentation techniques are necessary.

In the combinator reduction example, atoms represent the major tasks in the interpretive reducer: graph traversal in search of reducible combinators and their arguments, and graph transformations representing combinator reductions. This choice leads to large behavior descriptions, which limit the size of computations that the VCR can handle. Increasing the grain size of the atoms by combining several atoms eliminates details of individual combinator reductions; it is advantageous if it preserves the information needed for concurrency analysis and simulation.

Atoms in the parsing example represent parse tree construction; the attempt to build a tree node (setting a goal), the successful building of a node (goal resolution), the failure to build the node, and backtracking through previously built nodes, dismantling them.

Atom duration is measured by profiling the annotated program. Using an alternate expansion of the annotations, an application is converted into a profiler with code to

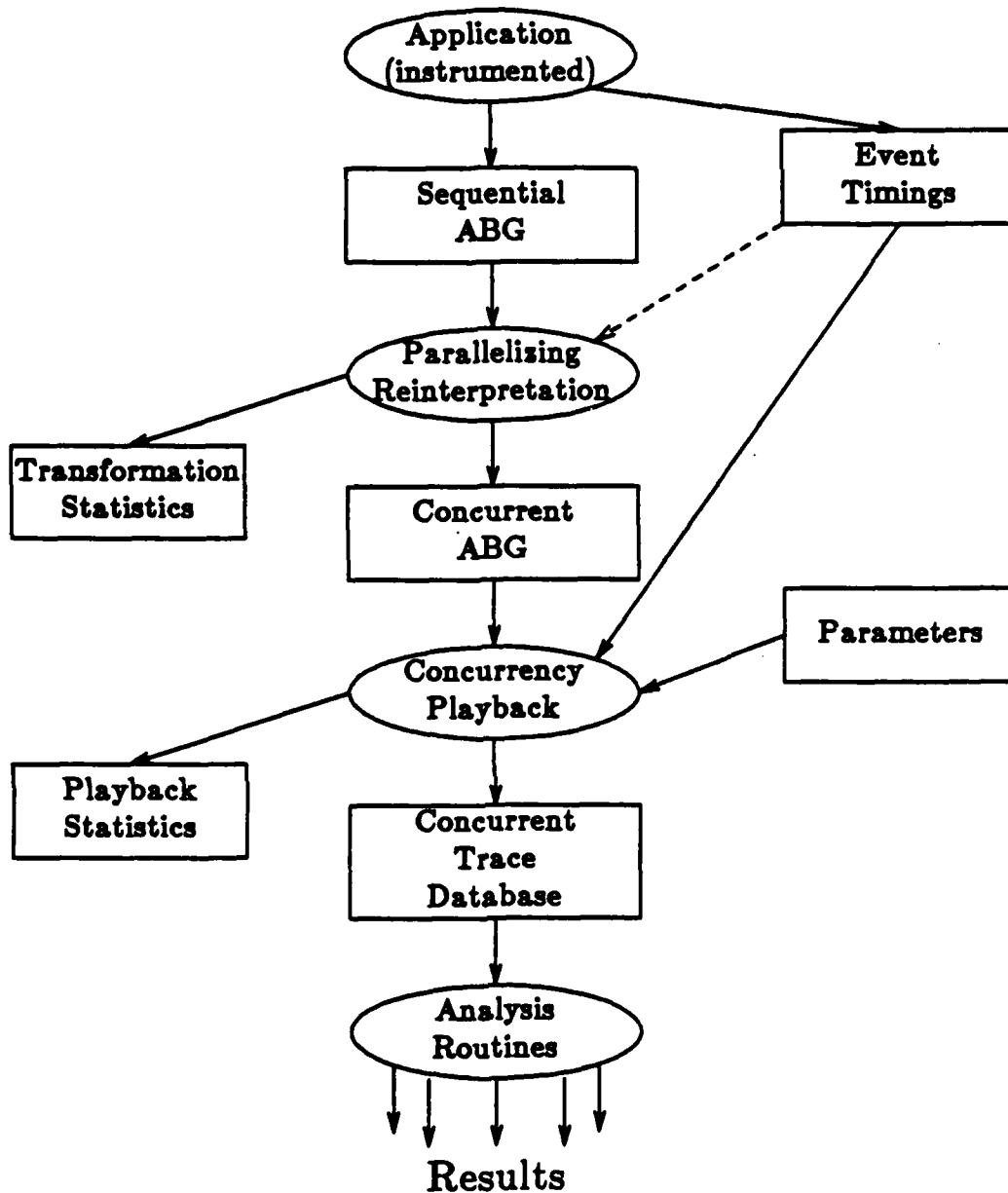


Figure 6-1: VCR Structure and Flow

count the number of system clock ticks since the occurrence of the previous annotation. The concept is similar to the Unix *monitor* and *profil* facility, but organized around the atoms instead of procedures. Other methods for measuring atom duration are possible. By separating the atom duration timing process from recording, the VCR methodology minimizes the interaction of the two (i.e., timing is not affected by recorder overheads), and provides flexibility in experimenting with timing parameters.

Timing data are stored in a separate table and loaded during playback initialization. Sequential recordings are transformed into ABG format for playback. Relevant attributes contained in the recording appear in each node: fields specifying the possible successor of the node, a duration field serving as an index into the timing table or as a direct duration value, and several application-dependent fields not directly used by the playback.

6.4.2. Playback

The term *playback* denotes an interpreter that instantiates the time of occurrence of the atoms in a behavior description. The concurrency playback is a discrete event simulation tool designed to simulate shared memory multiprocessor systems. It is written flexibly in C, and allows the insertion of policy modules to implement or simulate policies for spawning, scheduling, etc. The design is sufficiently general to accept different sets of atoms. To minimise playback changes for different atom sets, behavior classes distinguish different behaviors which the playback supports. Table 6-1 describes the current classes, which may, of course, be extended.

A table defines the atom set and specifies the behavior class of each atom in the behavior description.

To play back the behavior of a computation, the user specifies several input parameters to the playback. These include

- The concurrent behavior description, which may include spawn atoms to direct the the playback simulation of concurrency
- the atom set specification, contained in an external table
- a timing table, specifying the duration of the atoms
- the process startup overhead

sequential	simulate the next node of the graph
evaluation	select the branch as the next node; stack the next node to be resumed after return
spawn	initiate a process for the branch while the current process continues on the main branch
return	leaf of a branch reached: continue at the stacked root of the branch; end process if stack empty
terminate	terminate the current process

Table 6-1: Atom Behavior Classes

- the number of processors available, which may be unlimited

By providing an external timing table, a single behavior description can be simulated under many different timing models, allowing rapid evaluation of the effect of trade-off decisions in the design of a potential target system. It is thereby possible to determine quickly where optimizations will be effective, without having to implement them.

The raw output of the concurrency playback is a timestamped trace of atoms simulated, interleaving the concurrent activities. Since the output is a behavior description, the playback is a reinterpretation. The output trace can subsequently be reinterpreted for further instantiation or for analysis. The playback also produces a summary report of the simulated behavior.

6.4.3. Reinterpretations

The reinterpretation phase imposes new meanings on a behavior recording. The principal motivation for this phase is the application of concurrency policies to a sequential behavior description. This permits prediction of parallel execution behavior without having the computation run on a parallel system, and permits experimentation of many different concurrency policies on a single behavior description.

For generality, the granularity of the atoms is often finer than needed for concurrency experimentation. Reinterpretations permit compression of nonessential detail while preserving correct timing.

Another use of reinterpretations is identification of atom sequences which are not easily recognizable in the recording. A reinterpretation program may insert new atoms, with zero duration, which bracket a significant sequence of atoms, or insert an atom identifying the occurrence of an interesting event.

At times the atom set in a recording may not be fine-grained enough to provide the necessary behavior for an experiment. Reinterpretation can expand an atom into sub-atoms which show its fine structure.

Specific reinterpretations that have been implemented include

- An ABG builder, which transforms a linear trace into the partially-ordered directed graph ABG form. Typically, an instance of the ABG builder is necessary for each model of computation.
- An ABG sequentializer, to transform an ABG into a linear trace. This is the precise inverse of the ABG builder, allowing the user to transform the behavior description freely from one representation to the other. With the addition of a clock to allow timestamping (instantiation of the time-of-occurrence attribute), the sequentializer becomes a sequential playback.
- Concurrency inserter, to transform a sequentially-based behavior description to a concurrent behavior description. This has been done for each of the example domains. In each case, the reinterpretation is done as a filter on the sequential trace; combining it with the ABG sequentializer and the ABG builder provides an

ABG-to-ABG reinterpretation.

- For NL parsing, OR-parallel Prolog goals are pursued concurrently. A program written in C filters the sequential atom trace for a parse computation, inserting spawn atoms and removing atoms for backtracking, which is not needed in the concurrent model.
- For combinator-graph reduction, a transformation on the linear trace inserts spawns for all strict arguments after the first combinators with multiple strict arguments (such as most arithmetic operators).

6.4.4. Analysis

One representation of the behavior description generated by the concurrent playback is a text file which lists each atom encountered and its attributes. These include many of the originally recorded attributes as well as those instantiated by the playback, which include the time of occurrence, process number, processor number, etc. Analysis of this trace to determine various statistics is easily accomplished with a variety of Unix utilities, shell scripts, and small programs in sed, awk, or C.

As an example, the distribution of process lengths is obtained by extracting the start and completion atoms for all processes (with grep), sorting them by process number and secondarily by time (with sort), computing the process lengths by comparing the times for the pair of atoms for each procedure (a short awk program), and assigning the resulting numbers into ranges and graphing the counts as bar charts (Figure 6-2). A graph of active processes as a function of time (Figure 6-3) is obtained by extracting the same set of atoms (grep), sorting strictly by time (sort), and filtering through a program that maintains a count of active processes and graphs it. Figure 6-3 shows the processor-limited case, with values above the processor limit of 35 showing the number of queued processes.

Other analyses show the distribution of the processor time by process length (Figure 6-4) and speed-up (Figure 6-5). The speed-up analysis involves replaying the behavior of a single computation with a range of values for the number of processors (abscissa) and for the overhead incurred in starting a process (separate curves), collecting the times for each, and graphing the ratio to the sequential single-processor time. All the graphs shown are for one case of the parallel parsing study.

6.5. Conclusion

The VCR has been used in two distinct research efforts, one of which was not anticipated in its design. The results have been instructive, both as they confirm the validity and effectiveness of the approach and as they point to limitations of the system and potential improvements.

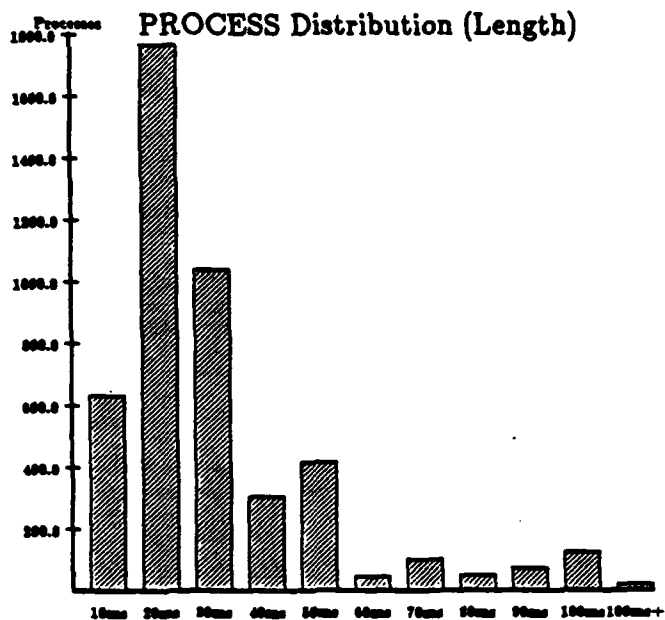


Figure 6-2: Typical Process Length Distribution.

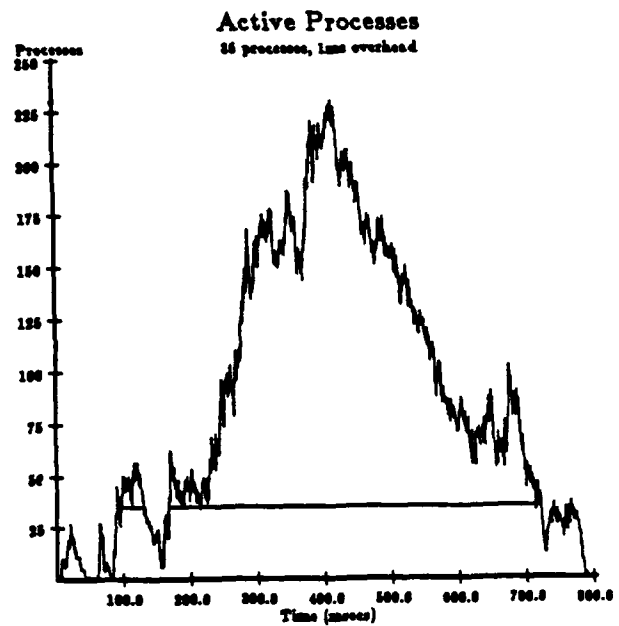


Figure 6-3: Typical Constrained Concurrency Graph.

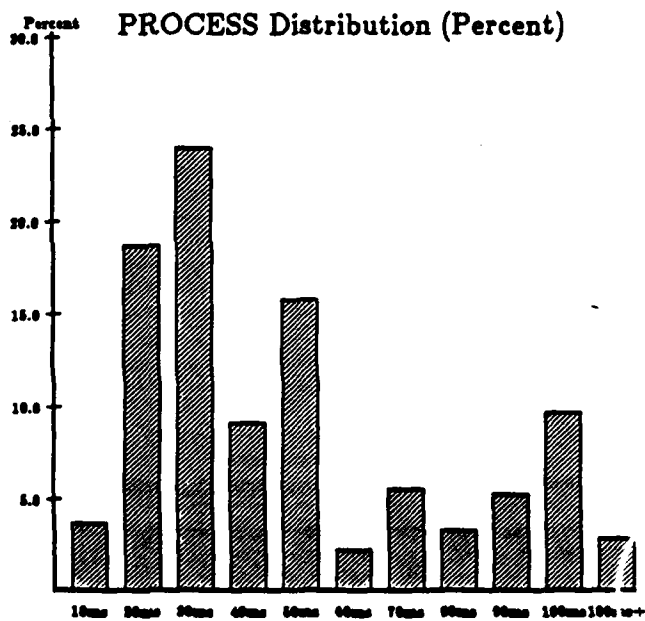


Figure 6-4: Typical Process Length Profile.

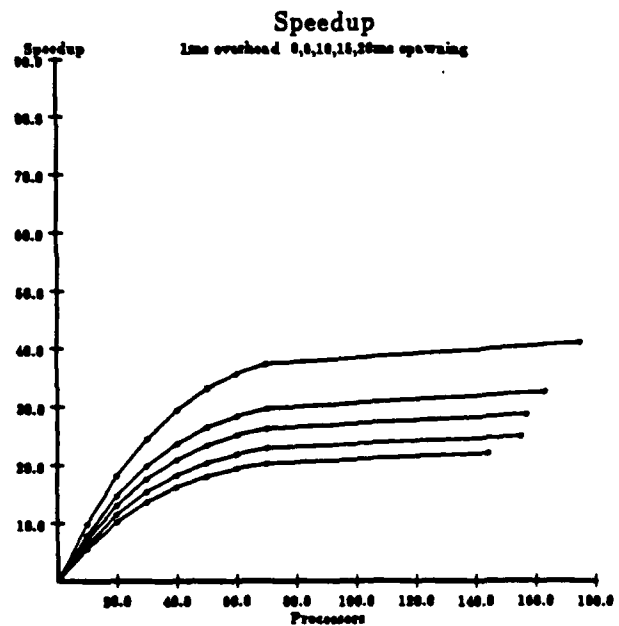


Figure 6-5: Typical Speed-up Graph.

6.5.1. Benefits of the approach

The principal benefits of VCR-based experimentation are the ease and speed of implementation of a wide variety of experiments and the speed of simulation.

Ease of Experiment Implementation

The first advantage is the result of basing the work on a strong model, which allows the researcher the luxury of working within a well-defined and well-supported experimental environment. An experimental scheduling policy, for instance, can be directly implemented as a policy module within the playback and be running in a matter of hours. Similarly, experiments on compile-time concurrency policies can be implemented in terms of existing atoms with a re-interpretation in short order. This compares favorably with the effort to produce a "real" implementation of the experimental features for evaluation, particularly since implementation details can obscure the effect of an experiment. The VCR allows an idealized implementation to be simulated to determine an upper bound for the effect of the feature, applying more realistic costs as they are estimated or measured.

As an example, in the NL parsing simulations, one initially unknown parameter was the cost of copying necessary state in spawning a process. Initial simulations imposed no cost on this operation, allowing an upper bound on the concurrency to be established. The effect of copying time was explored by ascribing different times to the spawn operation; Figure 6-5 shows the effect of times of 0, 6, 10, 15, and 20 milliseconds. Experiments on the parser measured the actual Prolog copying times for typical data structures, which were then incorporated in the final simulations.

Speed of simulation

The second advantage rests on two characteristics of the VCR: that many simulation parameters can be varied quite late in the methodology flow; and that the simulation of behavior is quite fast, typically much faster than running the instrumented program. All 45 of the data points for Figure 6-5 were run from the same behavior description, varying only atom durations and playback parameters. The relatively time-consuming process of generating the description, which took about an hour and a half, was performed only once. Each of the 45 points took 5 to 6 minutes to playback, and each graph took about 10 minutes to generate. For contrast, the parse itself, running on Quintus PROLOG, took about a minute. All times are for a diskless Sun 3/160.

6.5.2. Limits of the approach

The VCR approach is not without liabilities and limitations. These fall into two categories: those inherent to the VCR and those deriving from specific application concurrency models.

VCR limitations

The VCR is limited in the size of behavior graphs that it can manipulate, mostly by disk file sizes. We have managed to run cases where the size of the ABG was on the order of three million atoms, which translates into a disk size of about 60 megabytes. While the programs that manipulate the ABG support arbitrarily large structures, using an explicit virtual memory, the performance of these programs becomes seriously degraded at about the same point that disk space becomes excessive. For fine-grained atom sets, this is a serious restriction; the use of reinterpretations to reduce unneeded detail in the atom sets is one way of alleviating it.

A more important limitation is the form of interaction between atoms that the VCR allows. The general model assumes that concurrent activities do not interact very much; both instances of the VCR have limited and well-defined forms of interprocess interaction (in the parsing case, none except spawning) that do not require the simulation of detailed interactions. The limits of the system are such that simulation of rendezvous-level interactions in Ada, for instance, are feasible, while simulation of software lock interactions on a shared memory multiprocessor would probably be limited in scope. Interactions of this sort require that the atom set define each of the choice points in the interaction, which quickly expands the size of the behavior description beyond the limits of feasibility.

Model limitations

The most important limitation imposed by the model of an application domain is that the experimentation cannot easily go beyond the model. The model design must anticipate the questions that will arise during the research, and the directions it will take; aspects of the program behavior that are not explicitly in the model will not be available for inspection with the VCR. While it is always possible to amend the model, that usually requires recreating all the behavior descriptions and simulation results.

In some cases, a hierarchy of models is appropriate: for instance, one dealing with gross process interactions, one with detailed process interaction protocols, and a third dealing with memory allocation and garbage collection. With careful design of all models to be used, it is possible to go from one level to a more detailed level. This would, for instance, allow introduction of memory allocation demands to a behavior description defined in terms of larger atoms; this is possible if their detailed behavior can be predicted. It would then be possible to record at the middle level of detail and transform the recording to either of the others.

6.5.3. Directions for future work

On the conceptual end, the VCR is limited by the model of graph reduction from which it drew its inspiration. At some point we would like to generalize the VCR to allow simulation of more conventional systems. This will require some as-yet unknown insight to avoid the explosion of detail mentioned above.

Several areas of substantive improvement of the VCR implementation remain. We would like to see a more systematic way of specifying reinterpretations, and a more automatic way of keeping the atom sets and behavior descriptions linked. A file migration system to allow automatic archiving of large files, under pressure of disk space demand, would improve the day-to-day running of the system.

Acknowledgements

Many of the initial ideas for the VCR developed during discussions with Tom Blenko; Don McKay provided insights from a related concept he had previously defined. Clarke Arnold and Joel Coltoff contributed mightily to the implementation.

6.6. References

- 1 D. A. Turner, A new implementation technique for applicative languages. *Software Practice and Experience* 9, 1979, pp. 31-49.
- 2 P. Hudak and B. Goldberg, Serial combinators: "optimal" grains of parallelism. In *Functional Programming Languages and Computer Architecture*, Springer-Verlag LNCS 201, September 1985, pp. 382-388.
- 3 R.J.M. Hughes, Super-combinators. In *Proc. 1982 Symposium on Lisp and Functional Programming*, Pittsburgh, PA, August, 1982.
- 4 M. Scheevel, NORMA: A graph reduction processor. In *Proc. 1986 Conf. on LISP and Functional Programming*, ACM, August 1986, pp. 212-219.
- 5 L. Hirschman and K. Puder, Restriction Grammar: A Prolog Implementation. In *Logic Programming and its Applications*, D.H.D. Warren and M. VanCaneghem (ed.), 1985, pp. 244-261.
- 6 K. So, A.S. Bolmarcich, F. Darema, and V.A. Norton, A speed-up analyzer for parallel programs. In *Proceedings of the 1987 International Conference on Parallel Processing*, 1987.
- 7 T. J. LeBlanc and J. M. Mellor-Crummey, Debugging parallel programs with Instant Replay. *IEEE Trans. Computer C-36*(4), April 1987.
- 8 P. Bates and J. Wileden, High-level debugging of distributed systems: The behavioral abstraction approach, COINS 83-29, Dept. of Computer and Information Science, Univ. Massachusetts, 1983.

Section 7

The Virtual Computation Recorder

by

**Robert C. Smith
Joel B. Coltoff
William C. Hopkins**

Table of Contents

Section	Title	Page
7.1	Introduction	7-1
7.2	Recording	7-2
7.2.1	Juniper	7-3
7.2.2	PUNDIT	7-7
7.3	Reinterpretations	7-9
7.3.1	Juniper-B1	7-9
7.3.2	PUNDIT	7-12
7.3.3	Graph Building Reinterpreter	7-13
7.4	Playback	7-16
7.5	References	7-22
Appendices		
7-A	Atoms for Juniper-B1	7-23
7-B	PUNDIT Spawn Insertion Algorithm	7-24
7-C	Playback Trace	7-26

Table of Figures

Figure	Title	Page
7-1	VCR Flow Diagram	7-1
7-2	Spawn Insertion for Juniper-B1	7-10
7-3	Concurrent Execution in Juniper-B1	7-11
7-4	Juniper-B1 Compression	7-12
7-5	ABG Node Traversal	7-14
7-6	Interactive Screen Format	7-18
7-7	Evaluation of Shared Sub-Expression	7-21

Appendices

7-A	Atoms for Juniper-B1	7-23
7-B	PUNDIT Spawn Insertion Algorithm	7-24
7-C	Playback Trace	7-26

Table of Tables

Table	Title	Page
7-1	Atom Fields	7-3
7-2	Juniper-B1 Atom Record Format	7-6
7-3	Atoms in PUNDIT Recording	7-7
7-4	PUNDIT Recording Format	7-7
7-5	Types for PUNDIT Atoms	7-8
7-6	Combinators with Multiple Strict Arguments	7-10
7-7	Classification of Graph Node Edges	7-14
7-8	Juniper-B1 Abstract Behavior Graph Format	7-16
7-9	PUNDIT Abstract Behavior Graph Format	7-16
7-10	Batch Version Command Line Arguments	7-17
7-11	PUNDIT Atoms	7-17
7-12	<i>pundit_trace</i> File Format	7-19
7-13	Evaluation Branch States	7-21

Section 7

The Virtual Computation Recorder

7.1. Introduction

The Virtual Computation Recorder (VCR) is a simulation system providing information on concurrent program execution. The simulation system records a large application's sequential execution behavior, applies concurrency models to that behavior description, and produces a concurrent behavior description which when used to drive a discrete event simulator produces detailed information on concurrent execution. Figure 7-1 illustrates the basic components of the VCR. The results of the simulation provide information about the concurrency model's effectiveness for multiprocessor shared memory architectures. We provide a technical description of the VCR and the application of the VCR technology to two application domains, a concurrent programming system and a natural language parsing system.

The VCR does not specify the method of creating a recording other than define the recording format. In the two application domains discussed in this report, we annotated interpreters to produce the sequential execution behavior description. For tractable simulations, careful selection of the annotation points is necessary, with concurrency models specifying the level of detail needed in the recording. One must capture enough information to include all concurrency of interest, while hiding unnecessary detail. Although fine-grained execution detail should be hidden in the simulation, the simulation must account for the time spent executing the fine-grained detail to produce accurate results.

The recording consists of *atoms* representing the smallest unit of execution. Each atom is assigned an execution duration representing the execution time of the detail hidden in the atoms. Values for atom duration may be obtained by whatever means suitable; in our experiments we obtained values by profiling an instrumented sequential interpreter (in a manner similar to UNIX *gprof*).

Once a sequential recording is made, a series of *reinterpretation* programs transform the sequential execution description into a concurrent description manipulable by the simulator. These reinterpretation programs identify concurrent execution points in the sequential description according to the concurrency model under test, and transform a sequential description into a concurrent description (the *concurrent*

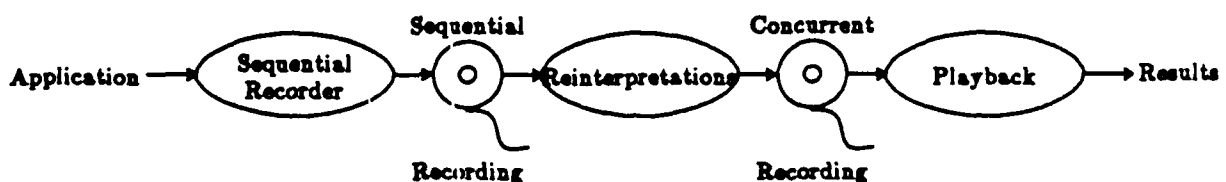


Figure 7-1: VCR Flow Diagram

recording).

The concurrent recording drives the simulator or *playback* device which simulates a multiprocessor shared memory system. The number of processors available to a program, a runtime parameter, may be unlimited (i.e. a processor is always available when requested) or finite (≥ 1). The playback produces a simulation summary and a timestamped trace of the concurrent recording. A series of programs summarizes the concurrent recording producing graphs showing the amount of concurrency over time, speedup over the sequential execution, and process length distributions. Additional playback parameters such as process startup overhead, copying costs associated with spawning processes, and atom duration tables permit a broad study of potential architectures.

The two application domains investigated were a concurrent programming system under development, called Juniper, which merges the functional and logic programming paradigms, and the PUNDIT¹ natural language processing system. Juniper is a programming language providing the properties for concurrent execution found in functional programming languages along with the logic programming constructs of Prolog-like languages. Of interest to us in PUNDIT was the amount and granularity of OR-parallelism in the grammar, and the speedups achievable through concurrent resolution of disjunctive goals, *or*.

Our investigation of Juniper proved the VCR concept and simulation system useful for simulating concurrency; since useful, large-grained concurrency models were not available, we report no results. After proving the VCR concept with Juniper, we applied the VCR to OR-parallelism in the PUNDIT parser, and produced significant results. We showed considerable OR-parallelism in natural language parsing systems; speedups in excess of 20 times for many sentences. Concurrency levels spiked to over 200 in some experiments with sustained concurrency of 50 not uncommon; long tails of low concurrency at startup and finish reduce the overall speedup. Results showed a linear relationship of speedup to sentence length, and speedup corresponded with search focus, a measure of the parser's ability to quickly focus on the correct parse.

7.2. Recording

A recording lists the execution steps required to produce the results of a program. The VCR defines no recording methodology other than the recording must consist of 14 byte atoms stored in binary form as Table 7-1 describes.

The *<user defined>* field contains user defined subfields with information important in identifying concurrency, and differs for each application. Information in the *<user defined>* field which helps relate concurrent execution to the sequential execution may be useful in interpreting the simulation results.

To provide flexibility in simulation, the VCR maintains timing data in a separate file; allowing the use of different timing data with the same recording (perhaps to

¹Prolog Understands Integrated Text

Field	Size (bits)
<i><user defined></i>	82
<i>atom</i>	10
<i>time-flag</i>	1
<i>time</i>	19

Table 7-1: Atom Fields

reflect a different processor). The *time* field in the recording contains an index into the timing table, but at times the timing data may not be easily kept in a separate file, so the *time-flag* field in the recording specifies whether the *time* field is an index or actual time value. The timing table is an ASCII file of records terminated with newlines; each record contains a time for a given atom. The first record contains the size of the timing table.

For reusability of the playback, the set of atoms defined for the application has a table of actions which specifies how to simulate an atom, primarily selection of the next atom. The playback section(4) of this report defines the set of actions. The only restriction on the recording method is that the recording expose all possible concurrency points in a given concurrency model. In our problem domains we chose to annotate sequential interpreters to produce the recordings. In choosing the annotation points, one needs to annotate all paths through the program and capture all points where concurrency may occur. These annotated interpreters are slow, so an essential feature of the VCR is a single recording of an application containing sufficient detail to produce concurrent recordings for any model desired. This means that in annotating the interpreter one should anticipate all possible points of concurrency for all models under consideration; otherwise, the interpreter must be re-annotated and all recordings recreated for each model, eliminating the advantage of using a single recording for testing many different models. The principle motivation of the VCR is to produce a single recording of an application's behavior (very time consuming), and to play back this recording under many different models (relatively fast).

The next two sections describe in detail the annotations used in the two applications under test.

7.2.1. Juniper

Juniper is a functional programming language based upon SASL [1] which compiles to SKI combinators [2] and includes logic variables. We used an early version of Juniper called Juniper-B1 [3]. The compiled code is represented as a binary graph of operators and associated arguments. An interpreter traverses the graph following left-most branches in search of operators which are transformed (reduced) into new operators with the reductions eventually reducing the graph to a node representing the solution, if one exists. Because it can be shown that combinator based programs exhibit referential transparency, combinator based programs are good candidates for

concurrent execution. The concurrency model first applied to Juniper applications was to spawn concurrent processes whenever a combinator had two or more strict arguments; strict arguments must be reduced to a value before combinator reduction proceeds (e.g. the *plus* operator must evaluate its operands before being reduced to the sum). Any recording based on this concurrency model should contain sufficient detail to identify combinator reductions and associated strict argument evaluation.

We annotated the evaluation of all combinators and their arguments. In addition, we annotated all significant paths through the interpreter to preserve timing characteristics. This meant annotating all branch points in the C code (*if-then-else*) if the different branches differed in length substantially. Any recursive function calls were bracketed by annotations, and any *for-while* loops were bracketed by annotations. This greatly increased the number of atoms in the recording, but allowed for accurate measurement of atoms. Some paths representing error conditions or IO were not annotated because, in general, these paths would not be traversed in the applications of interest (programs would work correctly, and IO would be infrequent). We compromised the accuracy of the recording to reduce the recording size by removing annotations around some very short loops and conditionals where alternatives were about the same length. We felt the improved simulation performance and disk usage reductions outweighed the slight inaccuracies in concurrency simulation. Appendix 7-A lists the atoms representing the annotations in the Juniper-B1 interpreter. Most of the atoms represent paths through the logic programming part of the interpreter, which did not expose any concurrency for this model. The atoms merely reflect time spent in the logic interpreter. The logic programming annotations were inserted in anticipation of later models that would include concurrent logic execution. This illustrates the design philosophy of capturing all concurrency points for all models in a single recording.

For the concurrency model studied (evaluate multiple strict arguments concurrently), three atoms are of interest, BCR_E, CR_E and Eval_E. The BCR_E atom represents the attempt to reduce a combinator, and the CR_E atom represents the completion of a combinator reduction. The Eval_E atom represents the request to evaluate a strict argument. A simple example of the occurrence of these atoms would be in evaluating the expression $a + b$. The following entries would appear in the recording:

```

BCR_E +
.
.
.
Eval_E a
.
.
.
Return_E a
Eval_E b
.
.
.
Return_E b
.
.
.
CR_E +

```

The opportunity to spawn concurrent tasks for argument evaluation occurs after the BCR_E atom. In the above example, the evaluation of operands *a* and *b* can occur simultaneously by spawning the evaluation of *b* before beginning the evaluation of *a*. Another atom Return_E represents the completion of an operand evaluation. In the addition example, the first Return_E atom represents the completion of the evaluation of *a* and control returns to evaluating the + combinator. The remaining atoms do not expose any concurrency in this model, and are simulated sequentially just as in sequential execution.

To obtain a recording a program, written in Juniper-B1 source, is compiled by the Juniper-B1 compiler producing a graph of SKI combinators. The compiler passes the object code (a SKI combinator graph) to the annotated interpreter which executes the compiled program. As annotations are encountered, the interpreter writes a record of the atom to a disk file with the format described in Table 7-2.

Records are stored in binary form to conserve disk space. Not all atoms use every field. For example, only Eval_E atoms² use the branch node field which indicates two successor paths for the atom, one representing the evaluation of the combinator argument and the other representing the path following the completion of argument evaluation. The three fields Reads, Writes, and Makes contain the counts of SKI graph reads, writes and node creations occurring for that atom. These fields permit parameterization of memory access times, and had we chosen a distributed memory model permit simulation of remote memory accesses.

²Another atom USE_E also uses the branch, but it is a special case of Eval_E and in this case can be considered indistinguishable. See the reinterpretation section of this paper for a complete description of the atoms.

Field	Size	Description
SKI Node	24	Current node address when atom occurred
SKI Branch	24	Node address where an argument evaluation begins
Reads	8	Number of graph reads since previous atom
Writes	8	Number of graph writes since previous atom
Makes	8	Number of graph node creations since previous atom
Combinator	10	Current combinator name
Atom	10	Atom name
Time Flag	1	Flag specifying time field is actual or indexed
Time	19	Time value for the atom or an index into a table of times

Table 7-2: Juniper-B1 Atom Record Format

Atom duration is not determined during the creation of the recording. Instead the recording specifies an index into a table of durations. Separate duration tables permit simulation of different speed processors without requiring repetition of the lengthy recording process.

In Juniper-B1, a profiling interpreter calculates the atom durations. Each atom annotation serves as a collection point for clock ticks issued by the profiler, and the number of ticks recorded since the previous atom occurrence measures the duration of an atom.

The interpreter initiates a profiling process which interrupts the interpreter every clock tick (about 20 milliseconds on a Sun 3). An interrupt handler in the interpreter counts the clock ticks, and upon encountering an annotation, the interpreter records the count for that atom, and resets the counter. The interpreter calculates atom durations after interpreting the program using the following formula:

$$\text{atom duration} = \frac{\text{number of clock ticks}}{\text{number of occurrences}} \times \text{time per tick}$$

For large programs the interpreter must do some garbage collection which we do not include in our model. To avoid including garbage collection time in our measurements, special annotations around the garbage collection calls save the clock tick count prior to garbage collection and restore the count immediately upon exiting the garbage collector. This prevents an atom from being unfairly charged for a garbage collection.

We profile a different, but equivalent, version of the interpreter to avoid including the overhead of recording the atoms. The profiling version uses the annotations of the recording version with a compile time switch replacing the recording code with the profiling code. This assures that the profiling measures the same code recorded. The profiling version incurs only a small amount of overhead for handling counters.

7.2.2. PUNDIT

Considerable research has been done on OR-parallelism in logic programming with no agreement on the potential speed improvement. Natural language systems seemed a good application to explore OR-parallelism since there is a significant amount of grammatical disjunction (*or*) in natural language parsing, and sequential execution is slow. PUNDIT is a natural language processing system [4] written in Prolog and designed principally to process Navy casualty reports.

We began with a simple model of concurrency; spawn processes for disjunctive alternatives whenever encountered. This model reveals the maximum concurrency available, although many short processes produce some inefficiencies in the concurrent execution. We expect refinements in the model, limiting concurrency to alternatives more likely to consume large amounts of processing, to produce a more efficient model. A version of PUNDIT already existed which produced a trace of the goals attempted in parsing sentences. This information exposed the concurrency of interest to us, and this PUNDIT version was easily adapted to the VCR. We identified a small group of atoms representing the start and completion of goals and backtracking from goal failures. Table 7-3 shows the list of atoms.

The output of the PUNDIT recording is a file of 14 byte binary atoms with fields as described in Table 7-4.

A profiler, similar to Juniper-B1's profiler, calculates atom duration in the PUNDIT parser. The annotations in the PUNDIT parser serve as tick collection points in the profiler. Since a single atom, *Start_goal*, represents the attempt to resolve many different rules, which might vary greatly in time, we further distinguish collection of

Name	Action
Start_goal	attempt to resolve a goal
Complete_goal	successful completion of a goal
Fail_goal	unsuccessful completion of a goal
Backtrack	backtrack to a alternate rule of a disjunction

Table 7-3: Atoms in PUNDIT Recording

Field	Size	Description
level	24	current level of the parse tree
type	8	name of grammar rule being resolved
label	24	identifies a disjunction and its alternatives
rule	24	pointer to a table giving full grammar rule definition
event	10	atom name
tflag	1	flag indicating a actual time or index in the time field
time	19	contains the atom duration or index into table of durations

Table 7-4: PUNDIT Recording Format

times by the type field in the recording. For example, a *Start_goal* atom representing the attempt to parse a *subject* may have a different time associated with the occurrence of that instance of a *Start_goal* than the attempt to parse a *verb*. This is also done for *Fail_goal*, *Complete_goal* and *Backtrack*. Two fields distinguish atoms in a recording, the atom field and the type field. With this view one might say that there are over 400 atoms in PUNDIT, since there are over 100 different grammar rules (types), and each of the four primary atom's type fields may have one of the more than 100 different type field values. Table 7-5 lists the values for the type field.

adjadj	apos	appos	assert_frag
assertion	astg	atom	avar
be_aux	clshould	center	commaopt
compound	conj_wd	dp1	dp1pn
dp2	dp2pn	dp3	dp3pn
dpsn	dstg	eqtovo	fragment
howqastg	internal_punct	isr	la
lar	lar1	lcda	lit
ln	lamer	lnr	lnsr
lp	lpro	lpror	lq
lqnr	lqr	ltvr	lv
lvenr	lvingr	lvr	namestg
nnn	np_selection	npn	npos
nq	nqnvar	nsnwh	nstg
nstg_frag	nstgo	nsvingo	ntovo
null	null_aux	null_main	nullc
nulln	nullobj	nullwh	nvar
objbe	objbe_frag	object	objtovo
or	passobj	pn	pnn
pnpn	predicate	qn	qnpos
qpos	question	ra	rel_clause
restr	rn	rpro	rq
runon	rv	sa	sentence
snwh	sub0	sub1	sub7
subject	sven	svo	thats
tovo	tpos	tvo	veno
venpass	vingo	vo	vso_selection
vvar	wh	wh_question	whln
xor	yesnoq	zerocopula	

Table 7-5: Types for PUNDIT Atoms

7.3. Reinterpretations

Reinterpretations transform a recording by applying the constraints of a concurrent execution model to the sequential recording. The principle motivation for reinterpretations is to apply a concurrency model to the recording, and thus produce a behavior description of the program running concurrently. The playback itself is a reinterpretation of the recording which simulates concurrent execution based on the durations assigned to atoms and produces a timestamped record of the concurrent execution. Since the playback, by itself is such an important component of the VCR, a separate discussion appears in section 4.

We apply several types of reinterpretations apply to recordings.

- Spawn Insertion
- Graph Building
- Atom Compression
- Atom Insertion

Spawn insertion reinterpretations apply the chosen static concurrency model to the recording. This introduces a new atom, SPAWN_E, to the recording specifying the concurrent execution of a sequence of atoms. Graph building transforms the recording into an acyclic directed graph more easily manipulable by the playback. Another reinterpretation program recognizes that many atoms exist only to preserve timing characteristics and are otherwise unnecessary in concurrent simulation. The atom compression program searches for threads of sequential execution and compresses the threads into a single atom whose duration is the sum of the atom durations in the thread. This improves playback performance and reduces demand for disk space. Occasionally, the set of atoms may not express an event of interest in the playback, or a particular sequence of atoms may be of interest, but are not readily apparent. Reinterpretation programs may insert atoms representing the occurrence of the event, and the playback record its occurrence recorded. The insertion of an atom generally should not affect the timing, so inserted atoms usually have no duration.

The following sections describe the reinterpretation programs used in the two application domains, Juniper-B1 and PUNDIT.

7.3.1. Juniper-B1

Two types of reinterpretation programs were developed for the Juniper-B1 studies, a spawn insertion program and an atom compression program. The spawn insertion program, representing the concurrency model under test, inserts SPAWN_E atoms at the start of combinator reduction for combinators with multiple strict arguments. The SPAWN_E atom directs the playback to simulate a process representing a strict argument evaluation. The parent process simulates the first strict argument evaluation, after spawning processes to simulate the remaining strict arguments. Most of these combinators are arithmetic combinators such as addition and subtraction (see Table 7-6). The reduction steps consist of a sequence of atoms bracketed by BCR_E(begin combinator reduction) and CR_E atoms(complete combinator reduction).

MATCH	2	EQ	2
GR	2	GRE	2
MUCHGR	2	MINUS	2
PLUS	2	TIMES	2
INTDIV	2	FDIV	2
MOD	2	POWER	2

Table 7-6: Combinators with Multiple Strict Arguments

Within that sequence two or more EVAL_E atoms indicate the demand for argument evaluation. The reinterpretation program locates the appropriate atom sequences, finds the EVAL_E atoms associated with that combinator, inserts the necessary SPAWN_E atoms immediately following the BCR_E atom, and inserts LABEL_E atoms immediately before the threads representing the argument evaluations. The graph building representation uses the LABEL_E atoms to link the SPAWN_E atoms to the appropriate subgraph representing the argument's evaluation, and removes the LABEL_E atoms. Figure 7-2 illustrates the spawn insertion for the addition combinator.

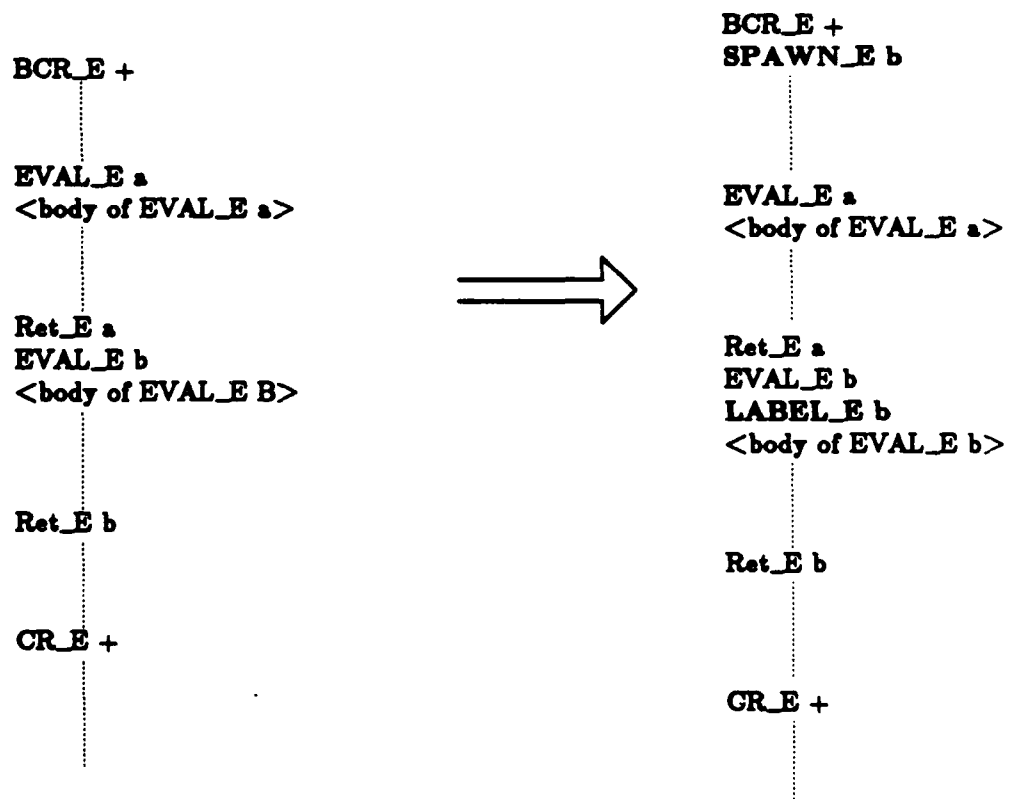


Figure 7-2: Spawn Insertion for Juniper-B1

The concurrent execution trace would show a process (1) (see Figure 7-3) encountering the BCR_E + atom, spawning process 2 to evaluate b, evaluation of a, and then attempting to evaluate b at which point process 1 would recognize that the value of b was available (process 2 had generated the result) or the evaluation of b was in progress. If in progress, process 1 would be suspended until process 2 had evaluated b when process 1 would be activated and complete the combinator reduction (CR_E +).

After applying the concurrency model (spawn insertion reinterpretation), many sequences or threads of atoms have no relevance in evaluating the concurrency model other than simulating the passage of time. These threads can be compressed into a single atom whose duration is the sum of the durations of all atoms in the thread. Compression improves simulation performance by reducing the number of events; performance and minimisation of simulation detail are a principle motivation for the VCR.

Sequences of atoms not representing the reduction of combinators with multiple strict arguments may be compressed into a single atom with the exception of sequences containing Eval_E and Ret_E atoms which bracket an argument evaluation. Since argument sharing occurs in Juniper-B1, these threads must not be compressed. The evaluation of a shared argument is not determined until playback when the first

Process 1

BCR_E +
SPAWN_E b

EVAL_E a
<body of EVAL_E a>

Ret_E a
EVAL_E b (wait)

CR_E +

Process 2

<body of EVAL_E b>

Ret_E b

Figure 7-3: Concurrent Execution in Juniper-B1

demand for the argument value causes the simulation of the thread Compression may occur within the boundaries of an Eval_E/Ret_E pair. Figure 7-4 illustrates the use of compression.

7.3.2. PUNDIT

For the OR-parallelism experiments with PUNDIT, we used three reinterpretation programs: a spawn reinterpreter identifies concurrency, a compression reinterpretation removes unnecessary atoms not needed in concurrent execution, and an atom insertion reinterpretation identifies successful parses.

The spawn insertion program finds the initial attempt to resolve a disjunction and inserts spawn atoms for all but the first alternative. This is done in two passes; the first identifies all the alternatives and the second inserts the SPAWN_E atoms. Label fields identify alternatives of a particular instance of a disjunction uniquely assigned by the recording interpreter to each disjunction encountered. The first pass of the insertion program identifies the disjunctions and associated alternatives by the label fields and builds an internal structure of pairs, labels and count of alternatives, for each disjunction. The second pass inserts $n-1$ SPAWN_E atoms (for a disjunction with n alternatives) immediately after the initial attempt to resolve the disjunction. In the simulation, the parent process resolves the first alternative, while the spawned processes resolve the remaining alternatives concurrently. The first pass of the spawn reinterpretation program inserts a LABEL_E atom marking the initial point of alternative resolution. There is a one-one correspondence between LABEL_E atoms and SPAWN_E atoms. To maintain this correspondence, a pair (anotated interpreter generated label, alternative number) links the SPAWN_E with the appropriate label record. To illustrate: consider the grammar rule $a ::= b;c;d$ which has alternatives b , c , and d as alternatives 0, 1, and 2. If the recording interpreter assigns label 1 to

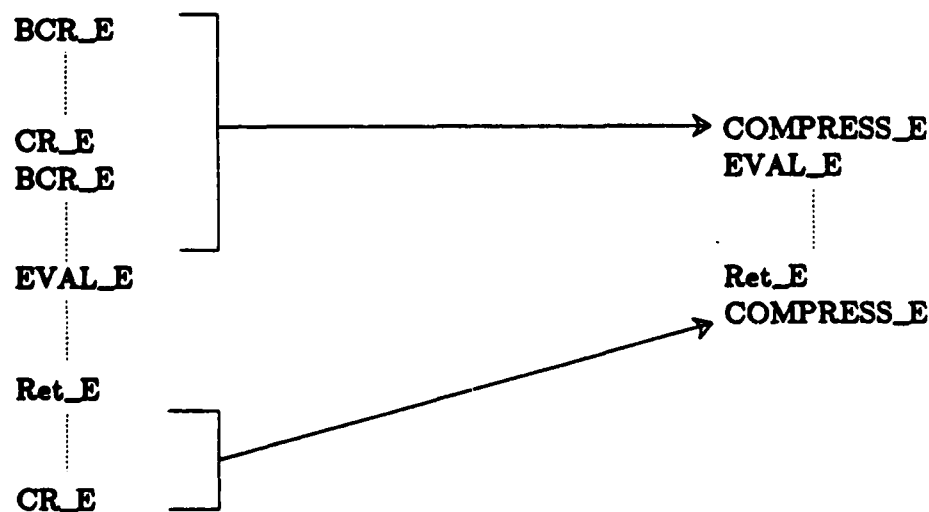


Figure 7-4: Juniper-B1 Compression

disjunction *a*, then the SPAWN_E atoms generated for alternatives *c* and *d* are assigned pairs (1,1) and (1,2), respectively.

The previous description is a very general description of the insertion of spawns. Disjunctions appear in several different forms in the recording causing variations in the insertions dependent on their form. Appendix 7-B shows the various forms of the disjunction and the transformations made to insert SPAWN_E and LABEL_E atoms in the recording.

Our initial concurrency model spawns processes to resolve all alternatives for every disjunction encountered except *xor* disjunctions which simulated sequentially. We call this model full spawning. Future experiments may select certain disjunctions (those with lengthy resolution times) for concurrent processing and resolve sequentially those with short resolution times. With a full spawning model, backtracking was not needed except in *xor* disjunctions. Since the sequential recording contains backtracking to resolve disjunction alternatives, these Backtrack_E and FAIL_E atoms must be removed in a fully spawned model. A compression reinterpretation program locates sequences of Backtrack_E and FAIL_E atoms representing backtracking to an alternative and compresses these sequences into a single atom, called a STOP_E atom. This replacement STOP_E atom signals the simulator to terminate the process (processes terminate upon failure). The duration for the STOP_E atom is the value of the first atom in the sequence of Backtrack_E-FAIL_E atoms. This differs from the Juniper compression (which sums the times of compressed atoms) since the backtracking is not done in the concurrent model.

Another reinterpretation program identifies successful parses since it is of interest to know when a successful parse tree is available for semantic analysis in the concurrent execution. The successful resolution of a goal (a Complete_goal atom) followed immediately by a Backtrack_E atom indicates a successful parse. The reinterpretation program finds these instances and replaces the Backtrack_E atom with two atoms, a Success_E atom and a STOP_E atom³. To preserve the correct timing characteristics, the reinterpretation program associates a duration of zero with success atoms, and uses the Backtrack_E atom's duration for the STOP_E atom duration.

7.3.3. Graph Building Reinterpreter

To provide an easily manipulable structure for the playback, the VCR requires input to the playback be an acyclic directed graph of atoms called an ABG or abstract behavior graph. The final reinterpretation program transforms recordings into the ABG format.

The graph consists of nodes representing the atoms and edges emanating from a node representing paths to successor atoms. A node may have more than one entering edge and zero, one or two exiting edges. The number of exiting edges is determined by the class of the node which is assigned to each atom by the simulation study designer. The class determines how the playback treats the exiting edges, and is

³In full spawning the additional parses have already been spawned, so no further processing is available.

described in Table 7-7. Most nodes have only a single entering edge except nodes with edges coming from Eval nodes which may have an arbitrary number. These multiple incoming edges occur when subgraphs are shared which occur in the Juniper-B1 applications (a full discussion on the handling of Eval nodes and shared subgraphs appears in the description of the playback).

Figure 7-5 illustrates the traversal by the playback of several classes of nodes. The sequential nodes have a single exiting edge and the playback merely follows the single edge. One can simulate sequential execution by skipping the concurrent model

Sequential	0	single edge
Eval	1	two edges; push second edge onto call stack; follow branch edge first; upon return follow second edge
Return	2	no edges; pop next edge from call stack
Use	3	same as Eval; used for shared values
Spawn	4	two edges; branch edge indicates initial atom of new process; second edge begins immediately as successor to parent process
Label	5	marker; not in the resulting ABG
Stop	6	no edges; terminate process do not pop call stack

Table 7-7: Classification of Graph Node Edges

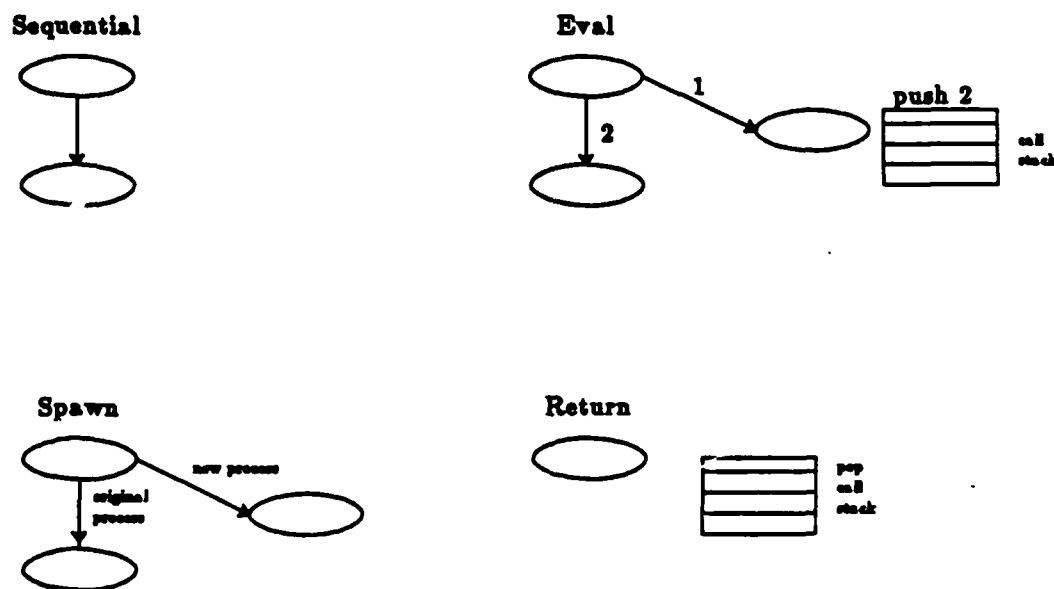


Figure 7-5: ABG Node Traversal

reinterpretation step and the graph builder would create a graph of only sequential nodes. The spawn node represents the identification of concurrent threads and contains two exiting edges. The first points to a subgraph containing the atoms for representing the new process, while the second points to the subgraph of atoms representing the continuing work of the original process. The Eval/Use nodes are treated like a function call, execute the body of the function and return upon completion to the step immediately after the call. One edge points to a subgraph representing the function and the second represents the continuation point after the first edge's subgraph has been traversed. The handling of these nodes in the playback uses a stack to save the second edge from Eval/Use nodes. Related to the Eval/Use nodes is the Return node which terminates the subgraph pointed to by an Eval/Use node. Return nodes have no exiting edges and signal the playback to pop the Eval stack to determine the next node. The Stop node (not pictured) has no exiting edges and signals the playback to terminate the process. It has the same effect as a Return node with an empty stack.

The format of the output is a binary file with record size the same for ABG's of PUNDIT and Juniper-B1 recordings, but the fields differ due to differences in the recordings. A compile time switch in the ABG builder and in the playback selects the correct record definitions for the recording. Table 7-8 and Table 7-9 describe the ABG formats for Juniper-B1 and PUNDIT. The ABG builder initializes the reserved field, used by the playback, to binary zeros. *successor* and *branch* fields represent the edge paths from the node and are integers specifying successor node numbers which are relative node addresses from the first record in the binary file (1 relative). Integer zero (0) in a successor or branch field indicates no edge.

The ABG builder resolves the edge paths (successor and branch) for each node in the input stream. Since the destination generally appears after the occurrence of a spawn node, branch addresses in a spawn must be determined later. The ABG builder maintains a linked list of unresolved spawn node addresses, and as it encounters destination nodes identified by label atoms, completes the spawn branch addresses and removes the entry from the linked list.

evaluation node branch addresses, although similar to *spawn* nodes, have branch nodes immediately following the evaluation node and successor nodes somewhat later in the input stream. The successor addresses then are not resolvable immediately, and the ABG builder uses a similar scheme for successor address resolution.

Because spawn and eval nodes must be kept in memory until completed, memory requirements for the ABG builder may exceed system limits for large recordings. To overcome this problem, the ABG manages its own graph memory to ensure sufficient memory. The ABG builder allocates graph nodes in large blocks (usually 1000 or 2000 node blocks), and writes completed blocks to a disk file freeing a block for new nodes. In the case when a new block is needed and no blocks can be written, incomplete blocks (needing only spawn/eval address resolution) may be written to the disk file and later read into memory to resolve an address. The ABG builder program begins using only the memory it needs and allocates additional memory as needed until no

successor	24	pointer to the successor node
read	8	count of SKI graph reads
branch	24	pointer to the branch node of an Eval or spawn
write	8	count of SKI graph writes
ski-node	24	the current position in the SKI graph
makes	8	count of the SKI graph node creations for this atom
reserved	16	fields used by the playback
event	10	the atom name
comb_name	10	the combinator name being reduced
time_flag	1	same as in recording
time	19	index into duration table or actual time if time_flag set

Table 7-8: Juniper-B1 Abstract Behavior Graph Format

successor	24	pointer to the successor node
branch	24	pointer to the thread representing an alternative evaluation
rule	24	index into a table containing the rule being resolved
level	24	current level in the parse tree
reserved	16	fields used by the playback
event	10	atom name
type	10	name of the rule being resolved
time_flag	1	set if time field contains an actual value
time	19	index into table of atom durations or actual duration value

Table 7-9: PUNDIT Abstract Behavior Graph Format

additional memory is available or until reaching a compiled-in upper bound on memory for graph nodes. This memory management scheme degrades performance of the graph builder somewhat, but does guarantee sufficient memory to build the ABG.

7.4. Playback

The final step in the VCR process is the reinterpretation of the ABG as a concurrently executing program on a shared memory multiprocessor. The playback, a discrete event simulation tool, accomplishes this task and produces a timestamped recording describing concurrent execution behavior. The playback permits many of the architectural parameters to be set at runtime, thus permitting a wide range of experiments, and its flexible design permits evaluation of many different applications even though the atoms in an ABG change from application to application.

There are two versions of the playback; a batch version for doing large numbers of experiments with a variety of parameter settings, and an interactive version using the standard Sun window system to permit limited observation of the simulation's progress.

The batch version provides much better performance, and most of the simulation runs used in producing the results shown later used the batch version. The batch version uses command line arguments to specify the experimental parameters (see Table 7-10).

The atom table file, the "-e <file-name>" command line option, specifies the file mapping atoms to node classes. Each application has its own atom table file. The file consists of records of ASCII strings followed by an integer; the strings being the atom name and the integer its class. In the ABG the atom name field of a node is represented as an integer which is derived from its position in the atom table file. Table 7-11 specifies that the START_GOAL_E atom is represented in the ABG as zero and a sequential node while COMPLETE_GOAL_E is represented as one and also a sequential node. See Table 7-7 in the ABG description for atom table class values.

START_GOAL_E	0
COMPLETE_GOAL_E	0
FAIL_GOAL_E	0
BACKTRACK_E	0
SPAWN_E	4
LABEL_E	5
STOP_E	6
DUMMY_E	1
SUCCESS_E	0

Table 7-11: PUNDIT Atoms

The playback selects the atom timing table by appending ".times" to the ABG file name, and assumes the location to be the current directory. The startup overhead and number of processors are optional arguments, with defaults of zero (0) microseconds startup overhead and unlimited processors.

With these runtime parameters, we can create many different architectural configurations including simulating different CPU's by using different timing tables for the same ABG. We have made use of the flexibility of timing files to avoid rerecording applications when we improved the atom duration measurement. In some instances changing the timing values requires reapplying the reinterpretations to the recording because some timing values are embedded into the ABG. This happens if a compression reinterpretation uses values from a timing table to determine the time for a compressed atom. In the PUNDIT application the concurrency reinterpretation

-e <file-name>	table of atoms file name
-a <file-name>	ABG file name
-s <number>	process startup overhead in microseconds
-p <number>	maximum available processors

Table 7-10: Batch Version Command Line Arguments

program embeds the cost (in terms of time) of spawning tasks (copying of unshared data) directly in the spawn nodes via a command line parameter. Any change in the spawning cost required rerunning the reinterperatation programs. The VCR design anticipated rerunning the reinterperatation programs and care was taken to ensure good performance, so the cost of rerunning the reinterperatation programs is small.

Execution of the interactive version creates a Sunview window consisting of subwindows with buttons and fields for selecting the simulation parameters and five strip chart subwindows for displaying atom occurrences. Simulation parameters are the ABG file name, the timing file name, the atom table file name, the process startup overhead, and the maximum number of available processors for the simulation run. In addition, a button is available to select up to five atoms for display on the strip charts. The strip charts show the occurrence of selected atoms as spikes on the strips. Each strip chart displays occurrences of a single atom during a simulation. This permits limited observation of the simulation's progress. Also, as an aid in observing the simulation's progress, the interactive playback continuously displays the count of atoms executed in the window. -6 shows the screen format schematically.

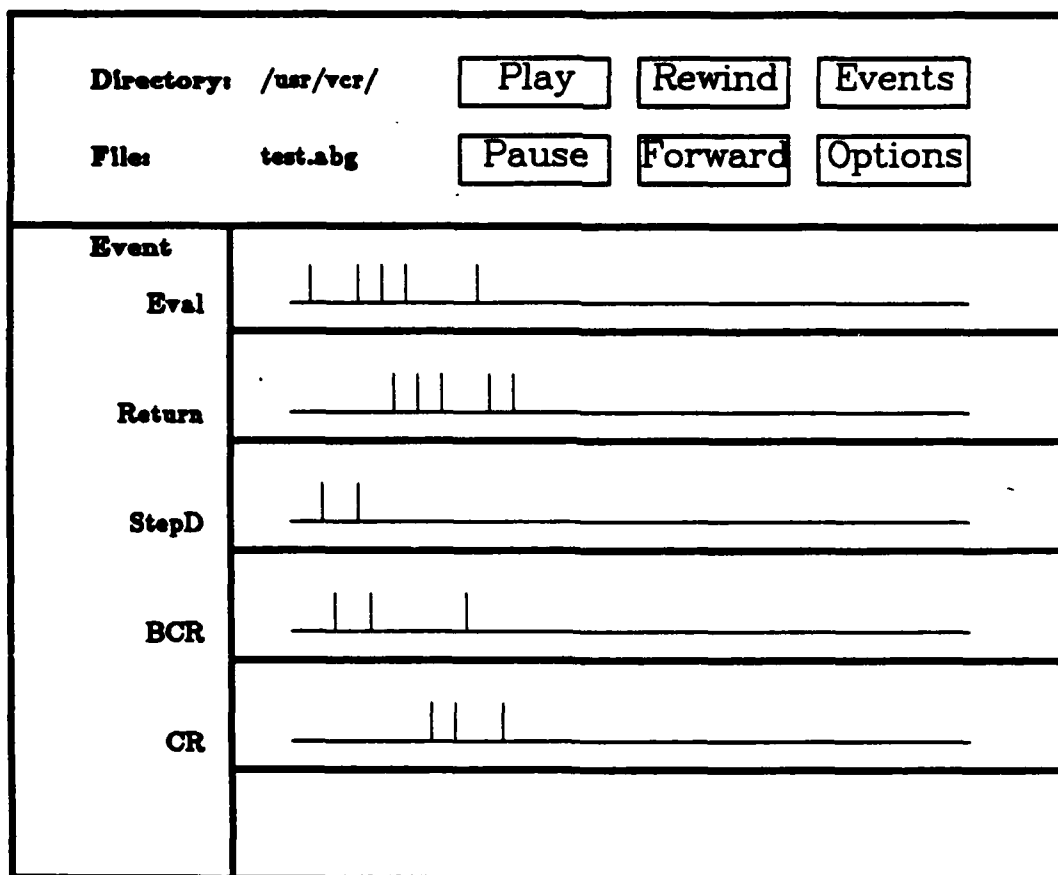


Figure 7-6: Interactive Screen Format

The playback produces two output files and a short summary of the run (written to standard output). One output file, *pundit_trace*, is a detailed description of the simulation run. The format is similar to an ASCII version of the recording but with additional atoms inserted to mark process initiation and termination, and all records contain atom completion times and process identification information. Table 7-12 shows the *pundit_trace* file format. The playback could produce a recording format file, as the annotated interpreters do, which could then be reinterpreted and played again by the playback. Currently we have found no immediate need for the recording format, and our interests are only in analyzing the output to judge the effectiveness of the concurrency policy. The second output file, and the information written to standard output summarise the simulation run giving the number of atoms simulated, concurrency rates(maximum and average), and process lifetimes (average, maximum and minimum). A sample fragment of the outputs appear in Appendix 7-C.

The playback uses a timeslicing mechanism to service all active unblocked processes. The timeslice length is a compile time option, and for our experiments represented 1 millisecond. Choosing a timeslice value too small can impact the playback's performance. A process table contains information about each process created during the simulation. For each timeslice all active processes assigned to a processor receive service. A process table entry with a terminated or blocked process receives no service. Each process has an ABG atom associated with it which represents the next atom to occur and a time when the atom occurs (time remaining). The playback decrements the atom's remaining time by the timeslice length until the duration reaches zero, at which time the playback triggers the event and executes actions defined by the atom. In most instances, this means selecting successor atoms, if any, and possibly spawning a new process. Once the playback determines a successor, the playback updates the process table entry with a new atom and calculates the time to trigger the event. If the timeslice length is less than the time remaining for an event occurrence, the difference is subtracted from the successor atom's occurrence time. More than one atom may be triggered in a single timeslice for a process if the durations of consecutive atoms are smaller than the timeslice length. The playback updates data on the number of active processes, the size of queues for processes blocked due to insufficient number of processors, and counters for the number of atoms

Atom (from ABG)
ABG node number
<user-defined fields> (from ABG)
atom duration (from ABG)
atom occurrence time
process number
processor number
process link

Table 7-12: *pundit_trace* File Format

occurring after each timeslice.

The playback handles five basic types of atoms, sequential single successor atoms, evaluation atoms with two successors, return atoms with no successors, terminate atoms with no successors, and spawn atoms with two successors, one a branch for a new process. The simulation of a sequential atom merely designates the single successor atom as the next atom simulated for that process.

The evaluation atoms specify simulation of the branch path first, and upon completion of the branch, simulation of the second successor path. This acts like a function call; execute the function first, and upon return execute what follows the function. To accomplish this the playback associates a call stack with each process to maintain a list of successors. The third atom type, the return atom, terminates a branch of the ABG causing the call stack to be popped to produce the process's next atom. Return atoms contain no successor atoms. An empty call stack causes process termination which may not mean an error under some models.

The evaluation and return node classes were needed to handle shared expressions in Juniper-B1 whose implementation permits sharing of common sub-expressions. The evaluation-return construct permits isolation of the shared sub-expression in the simulation. In a recording several evaluation nodes may have the same branch addresses to represent the sharing of a common subexpression. In the playback the first reference encountered simulates the sub-expression evaluation producing a result or value. Subsequent references need only retrieve the result. To simulate this situation properly a control mechanism in the playback locks the ABG subgraph from access by later references during the simulation of the sub-expression. Once the sub-expression simulation completes, the simulation of its return node, references to the sub-expression must simulate the retrieval of the result only. The control mechanism in the playback marks the head node of the ABG blocked when the first reference to the sub-expression begins simulation. The playback suspends any processes referring to this blocked node until the initial reference completes the sub-expression simulation. The return node simulation marks the block at the head of the sub-expression's graph complete, and any processes blocked on that sub-expression resume. For the Juniper-B1 application the retrieval of a previously computed result caused a small amount of work to be done creating a small subgraph in the ABG. Since we do not know which evaluations cause the sub-expression to be evaluated in a parallel system, we must provide all references to the sub-expression equal ability to evaluate or retrieve the result. We accomplish this by having all evaluations point to the subgraph representing sub-expression evaluation and the head of the sub-expression subgraph point to the subgraph representing result retrieval via the branch address field. In JuniperB1 the head of the sub-expression subgraph was guaranteed to be a sequential class node and thus had an unused branch address field. Table 7-13 summarizes the possible states of evaluation node branch addresses. Figure 7-7 illustrates the sharing of sub-expressions in an ABG.

The fourth atom type, the terminate or stop atom, directs the playback to terminate the current process. The playback marks the process in the playback process

unmarked	follow successor address
blocked	suspend task until result available
complete	follow subgraph head's branch field for result

Table 7-13: Evaluation Branch States

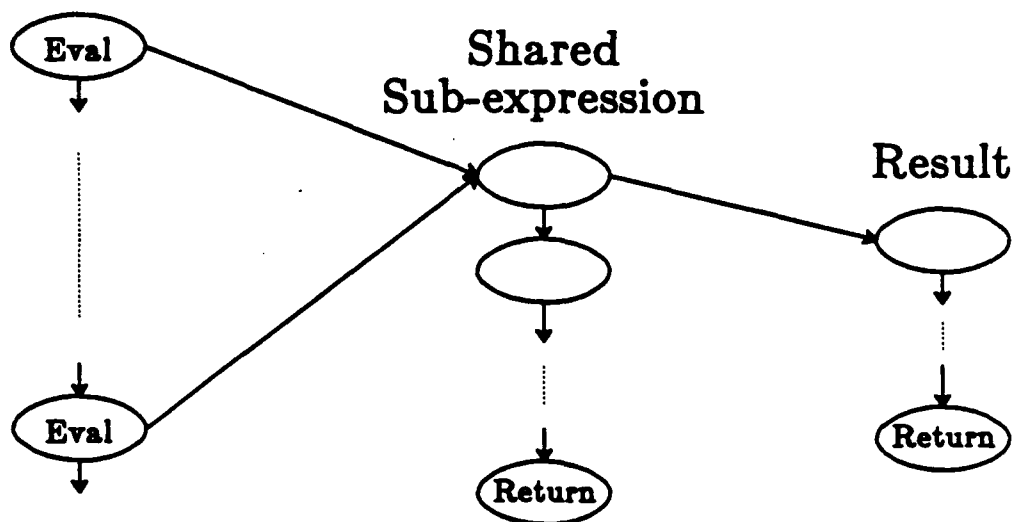


Figure 7-7: Evaluation of Shared Sub-Expression

table complete, frees the processor, and checks the process queue for a process to assign to the freed processor, otherwise the processor goes into an available pool. If the number of processors is unrestricted, the playback does not assign processors to processes, but only maintains an active process.

The fifth atom type, the spawn atom, provides the opportunities for concurrency. Spawn atoms contain two successor atoms, one selects the successor atom of the current process (the nodes successor field), and the second, the branch field, selects the initial atom for a new process. The playback creates a new process table entry, and if no processors are available, blocks the process and places it in a queue. If not blocked the playback schedules the start of the process based on the process startup overhead specified at runtime. If process startup is zero, the process startup begins immediately and the playback schedules the first node for that process.

When the playback has no more active processes and queues are empty, the simulation terminates. The playback produces the summary appearing in the .trc file.

7.5. References

- 1 D. A. Turner, *SASL Language Manual*. University of St. Andrews, 1976.
- 2 D. A. Turner, A new implementation technique for applicative languages. *Software Practice and Experience* 9, 1979, pp. 31-49.
- 3 T. Blenko, Definition of Juniper-B1, A functional + logical programming language, Technical Memo 78, Paoli Research Center, Unisys Corp., Paoli, PA, February, 1988.
- 4 L. Hirschman and K. Puder, Restriction Grammar: A Prolog Implementation. In *Logic Programming and its Applications*, D.H.D. Warren and M. VanCaneghem (ed.), 1985, pp. 244-261.

Appendix 7-A: Atoms for Juniper-B1

Atoms for Juniper-B1			
StepD_E	Eval_E	USE_E	Return_E
BCR_E	CR_E	Coas_BCR_E	Coas_CR_E
Fca_E	Fca_CR_E	B_RT_DB_E	B_EQ_E
B_CP_AS_GOAL_E	COPY_AS_GOAL_1_E	COPY_AS_GOAL_2_E	COPY_AS_GOAL_3_E
COPY_AS_GOAL_4_E	CALL_1_2_E	RESOLVE_1_E	RESOLVE_CONT_E
RESOLVE_APP_E	R_LOOKUP_2_E	R_LOOKUP_4_E	ID_1_E
HASH_DB_1_E	HASH_DB_2_E	RETRIEVE_DB_1_E	RETRIEVE_DB_2_E
RETRIEVE_DB_3_E	RETRIEVE_DB_4_E	PROVE_CONT_LOOP_E	CALL_1_E
POSTFIX_1_E	POSTFIX_2_E	AVL_INSERT_1_E	AVL_INSERT_2_E
AVL_INSERT_3_E	AVL_INSERT_4_E	AVL_INSERT_5_E	TOP_PROVE_LOOP_E
TOP_RESOLVE_LOOP_E	UNIFY_1_E	UNIFY_2_E	UNIFY_3_E
UNIFY_4_E	UNIFY_TERMS_1_E	UNIFY_TERMS_2_E	UNIFY_TERMS_3_E
UNIFY_TERMS_4_E	UNIFY_TERMS_5_E	UNIFY_TERMS_6_E	UNIFY_TERMS_7_E
UNIFY_TERMS_8_E	UNIFY_TERMS_9_E	UNIFY_TERMS_10_E	UNIFY_TERMS_11_E
UNIFY_TERMS_12_E	UNIFY_TERMS_13_E	UNIFY_TERMS_14_E	UNIFY_TERMS_15_E
UNIFY_TERMS_16_E	UNIFY_TERMS_17_E	UNIFY_TERMS_18_E	UNIFY_TERMS_19_E
UNIFY_TERMS_20_E	UNIFY_TERMS_21_E	UNIFY_TERMS_22_E	UNIFY_TERMS_23_E
UNIFY_TERMS_24_E	UNIFY_TERMS_25_E	UNIFY_TERMS_26_E	UNIFY_TERMS_27_E
UNIFY_TERMS_28_E	UNIFY_TERMS_29_E	UNIFY_TERMS_30_E	UNIFY_TERMS_31_E
UNIFY_TERMS_32_E	UNIFY_TERMS_33_E	UNIFY_TERMS_34_E	UNIFY_TERMS_35_E
UNIFY_TERMS_36_E	UNIFY_TERMS_37_E	UNIFY_TERMS_38_E	UNIFY_TERMS_39_E
UNIFY_TERMS_40_E	UNIFY_TERMS_41_E	UNIFY_TERMS_42_E	UNIFY_TERMS_43_E
UNIFY_TERMS_44_E	UNIFY_TERMS_45_E	UNIFY_TERMS_46_E	UNIFY_TERMS_47_E
UNIFY_TERMS_48_E	UNIFY_TERMS_49_E	UNIFY_TERMS_50_E	UNIFY_TERMS_51_E
UNIFY_TERMS_52_E	UNIFY_TERMS_53_E	UNIFY_TERMS_54_E	UNIFY_TERMS_55_E
UNIFY_TERMS_56_E	UNIFY_TERMS_57_E	UNIFY_TERMS_58_E	UNIFY_TERMS_59_E
UNIFY_TERMS_61_E	UNIFY_TERMS_62_E	UNIFY_TERMS_63_E	UNIFY_TERMS_64_E
UNIFY_TERMS_65_E	UNIFY_TERMS_66_E	UNIFY_TERMS_67_E	UNIFY_TERMS_68_E
UNIFY_TERMS_70_E	UNIFY_TERMS_71_E	UNIFY_TERMS_72_E	UNIFY_TERMS_73_E
UNIFY_TERMS_74_E	UNIFY_TERMS_75_E	UNIFY_TERMS_76_E	UNIFY_TERMS_77_E
UNIFY_TERMS_78_E	UNIFY_TERMS_79_E	UNIFY_TERMS_80_E	UNIFY_TERMS_81_E
UNIFY_TERMS_82_E	UNIFY_TERMS_83_E	UNIFY_TERMS_84_E	HANDLE_T2_AP_1_E
HANDLE_T2_AP_2_E	HANDLE_T2_AP_3_E	HANDLE_T2_AP_4_E	HANDLE_T2_AP_5_E
HANDLE_T2_AP_6_E	HANDLE_T2_AP_7_E	HANDLE_T2_AP_8_E	HANDLE_T2_AP_9_E
HANDLE_T2_AP_10_E	HANDLE_T2_AP_11_E	HANDLE_T2_AP_12_E	HANDLE_T2_AP_13_E
HANDLE_T2_AP_14_E	HANDLE_T2_AP_15_E	GROUND_1_E	GROUND_2_E
GROUND_3_E	GROUND_4_E	GROUND_5_E	GROUND_6_E
GROUND_7_E	GROUND_8_E	SEMI_GROUND_1_E	SEMI_GROUND_2_E
SEMI_GROUND_3_E	SEMI_GROUND_4_E	SEMI_GROUND_5_E	SEMI_GROUND_6_E
SEMI_GROUND_7_E	SEMI_GROUND_8_E	LOOKUP_1_E	LOOKUP_2_E
LOOKUP_3_E	LOOKUP_4_E	CANONICAL_1_E	CANONICAL_2_E
CANONICAL_3_E	CANONICAL_4_E	ULTIMATE_1_E	ULTIMATE_2_E
ULTIMATE_3_E	ULTIMATE_4_E	ULTIMATE_5_E	ULTIMATE_6_E
ULTIMATE_7_E	SEMI_ULTIMATE_1_E	SEMI_ULTIMATE_2_E	SEMI_ULTIMATE_3_E
SEMI_ULTIMATE_4_E	SEMI_ULTIMATE_5_E	SEMI_ULTIMATE_6_E	SEMI_ULTIMATE_7_E
SEMI_ULTIMATE_8_E	SEMI_ULTIMATE_9_E	SEMI_ULTIMATE_10_E	SEMI_ULTIMATE_11_E
UNIQ_ULTIMATE_1_E	UNIQ_ULTIMATE_2_E	UNIQ_ULTIMATE_3_E	UNIQ_ULTIMATE_4_E
UNIQ_ULTIMATE_5_E	UNIQ_ULTIMATE_6_E	APPEND_1_E	APPEND_2_E
APPEND_3_E	REVERSE_1_E	CONS_TO_GOAL_1_E	CONS_TO_GOAL_2_E
LOOKUP_VAR_1_E	LOOKUP_VAR_2_E	APPEND_CONT_E	SPAWN_E
LABEL_E	COMPRESS_E		

Appendix 7-B: PUNDIT Spawn Insertion Algorithm

Definition	Algorithm
rule ::= a;b;c	Start_goal 1 rule ::= a;b;c [Spawn (1,1)] [Spawn (1,2)] Start_goal 1 or a . . . [Label (1,1)] Start_goal 1 or b . . . [Label (1,2)] Start_goal 1 or c

Definition	Algorithm
rule ::= a rule ::= b	[Spawn (1,1)] Start_goal 1 rule ::= a . . . [Label (1,1)] Start_goal 1 rule ::= b

Definition	Algorithm
rule ::= a,(b;c;d;e)	Complete_goal a [Spawn (1,1)] [Spawn (1,2)] [Spawn (1,3)] Start_goal 1 or b . . . [Label (1,1)] Start_goal 1 or c;d;e Start_goal 1 or c . . . [Label (1,2)] Start_goal 1 or d;e Start_goal 1 or d . . . [Label (1,3)] Start_goal 1 or e

Appendix 7-C: Playback Trace

Sample Playback Trace										
time	addr	type	data	len	duration	channel time	pid	bits	parent	
START_GOAL_S	7040	0	memory	000000	27	0000	1010701	000	10	0
START_GOAL_S	7114	24	lv	000000	20	0000	1010700	000	10	0
START_GOAL_S	7210	72	cr	000000	20	0000	1010100	001	9	0
START_GOAL_S	7304	72	cr	100000	20	0000	1010004	000	7	0
START_PROCESS	7400	0	cdjcdj	0	0	0	1010000	000	11	071
SPAWN_S	7401	0	appos	100071	1	0000	1010000	071	14	000
START_PROCESS	1004	0	cdjcdj	0	0	0	1010700	000	17	070
SPAWN_S	1005	1	appos	00040	1	0000	1010700	070	20	007
START_GOAL_S	7004	72	cr	001007	20	0000	1010100	074	2	0
COMPLETE_GOAL_S	7707	72	cr	000007	10	0000	1010700	070	10	0
START_GOAL_S	6000	72	cr	000070	20	0000	1010700	000	0	0
COMPLETE_GOAL_S	7110	20	lv	000000	20	0000	1010000	007	10	0
START_GOAL_S	6000	72	cr	010040	20	0000	1010700	000	10	0
START_GOAL_S	6000	20	lv	100000	0	0000	1010407	070	0	0
START_GOAL_S	7007	72	cr	000000	20	0000	1010100	004	20	0
START_GOAL_S	6000	72	cr	007074	20	0000	1017700	000	4	0
COMPLETE_GOAL_S	7000	20	lv	100000	0	0000	1010700	070	0	0
START_GOAL_S	6000	72	cr	100707	20	0000	1010000	000	7	0
COMPLETE_GOAL_S	7110	07	svcr	000000	20	0000	1010000	007	10	0
START_GOAL_S	7000	20	lv	001000	27	0000	1010100	074	2	0
START_GOAL_S	7700	20	lv	000070	10	0000	1010700	070	10	0
START_PROCESS	0000	0	cdjcdj	0	0	0	1000701	000	0	000
SPAWN_S	0044	1	appos	000017	1	0000	1010701	000	10	000
START_GOAL_S	0004	0	ctsm	000001	27	0000	1010701	000	0	0
START_GOAL_S	0004	0	ctsm	010007	27	0000	1000700	000	10	0
COMPLETE_GOAL_S	0044	00	cr	100000	0	0000	1000007	070	0	0
START_PROCESS	7004	0	cdjcdj	0	0	0	1001100	000	10	001
SPAWN_S	7000	1	appos	007000	1	0000	1000100	001	0	000
STOP_S	0000	0	connect_flag	007000	27	000	1000004	000	0	0
END_PROCESS	0	0	cdjcdj	0	0	0	1000104	000	0	0
START_GOAL_S	7000	0	ctsm	000014	27	0000	1000100	004	20	0
STOP_S	7000	0	types	000010	27	000	1000011	004	20	0
END_PROCESS	0	0	cdjcdj	0	0	0	1000011	004	20	0
START_GOAL_S	0000	72	cr	100071	20	0000	1000004	000	11	0
START_GOAL_S	1004	72	cr	00040	20	0000	1000710	007	17	0
START_PROCESS	0007	0	cdjcdj	0	0	0	1000040	000	24	071
SPAWN_S	0000	0	connect_flag	100001	1	0000	1001040	071	14	000
START_PROCESS	1000	0	cdjcdj	0	0	0	1000700	001	21	070
SPAWN_S	1044	0	appos	02001	1	0000	1001700	070	20	001
START_GOAL_S	7007	04	svcr	007100	27	0000	1001700	000	4	0
COMPLETE_GOAL_S	7700	20	lv	000007	10	0000	1000000	070	10	0
ART_GOAL_S	0000	07	memory	100000	21	0000	1000010	000	7	0
OP_S	0000	0	appos	007100	27	000	1000170	000	4	0
END_PROCESS	0	0	cdjcdj	0	0	0	1000170	000	4	0
START_GOAL_S	7117	04	svcr	007000	20	0000	1000000	007	10	0
COMPLETE_GOAL_S	0000	0	ctsm	010000	27	0000	1000010	000	10	0
COMPLETE_GOAL_S	7700	11	connect_flag	000000	10	0000	1004000	070	10	0
START_GOAL_S	7001	00	lv	000000	27	0000	1004177	001	0	0
START_GOAL_S	0000	72	cr	100400	20	0000	1004000	000	11	0
START_GOAL_S	1004	72	cr	000007	20	0000	1004704	007	17	0
START_GOAL_S	0000	72	cr	000017	20	0000	1004700	000	0	0
COMPLETE_GOAL_S	7110	04	svcr	007070	20	0000	1000400	007	10	0
COMPLETE_GOAL_S	0000	72	cr	010001	20	0000	1000070	000	10	0
START_GOAL_S	0000	72	cr	110140	20	0000	1000004	071	14	0
START_PROCESS	7011	0	cdjcdj	0	0	0	1000100	000	1	074
SPAWN_S	7000	1	appos	000100	1	0000	1000100	074	0	000
START_PROCESS	0070	0	cdjcdj	0	0	0	1000701	000	20	000
SPAWN_S	0040	0	appos	000710	1	0000	1000701	000	10	000
START_GOAL_S	7004	20	lv	000700	27	0000	1000177	000	10	0

Sample Playback Summary File (.trc)

Event count

START_GOAL_E - 19416

COMPLETE_GOAL_E - 7731

FAIL_GOAL_E - 25

BACKTRACK_E - 18

SPAWN_E - 4332

STOP_E - 4333

DUMMY_E - 1

SUCCESS_E - 2

Time to execute graph is 4.04333e+06 microseconds

Execute time of all nodes is 1.22759e+08 microseconds

Number of processes spawned is 4333

Number of spawns encountered is 4332

Average time for a process is 28331.2

Shortest process is 8401

Longest process is 154858

Process startup overhead 1000

Section 8

Directions for Future Investigations

by

**William C. Hopkins
Principal Investigator**

Table of Contents

Section	Title	Page
8.1	Research Directions	8-1
8.1.1	Extension of Compilation Technology to Emerging Languages	8-1
8.1.2	Parallelism in Spoken Language Understanding	8-2
8.2	Tools and Environments	8-3
8.2.1	Programming Environment for Implicitly Parallel (Declarative) Languages	8-3
8.2.2	Ada tasking	8-4
8.3	References	8-4

Section 8

Directions for Future Investigations

We have identified two major directions for future research resulting from the MASC program, and several technology spin-off directions. The major research directions build on the program successes in compilation technology and application parallelism analysis, while the technology spin-offs are related to the simulation tools developed in the program.

8.1. Research Directions

The MASC program has made major progress in the compilation of logic, and in the identification and exploitation of parallelism in AI applications. This section describes new research directions that build on those accomplishments.

8.1.1. Extension of Compilation Technology to Emerging Languages

The gadget technology described in Section 3 of this report is applicable to a wide spectrum of languages which use logic variables and unification in a λ -calculus setting. Section 4 demonstrates that it can also be effectively integrated with a more traditional sequential machine for Prolog. Given this breadth of applicability, and the optimisation potential provided by the gadget approach for logic programs, there are a large number of directions that further research may take. By focusing on support of emerging languages, we expect to accomplish the following:

- merge the MASC research into the *mainstream of current research* in programming language design and implementation,
- result in useful implementations, thus presenting the greatest *leverage to the research community*,
- *concentrate implementation efforts to support current research* in programming language design, avoiding the sometimes problematical aspects of older languages that are not well-motivated.

By *emerging languages* we mean a class of languages that include both functional and logic programming concepts, are based on sound language design principles, and enjoy widespread interest in the research community. Two prominent candidates are λ -Prolog [1], and the *Common Prototyping Language* currently being defined by an informal group of researchers with DARPA encouragement. λ -Prolog has the advantages of several years' work by researchers at Penn and Duke, and a growing body of users, including the Formal Methods Branch of the Unisys West Coast Research Center, the ERGO project at Carnegie Mellon, and others. We have established contact with the Common Prototyping Language group, and will integrate that language

design into our thinking when their report is released later in 1988. The rest of this discussion, while framed in terms of λ -Prolog, should be understood to apply to this broader language class.

A common thread of these languages is the use, in the framework of the typed λ -calculus, of *higher-order logic*, which means, strictly, allowing logic variables to represent logical relations. More generally, it allows programs to talk and reason about programs in a natural way, a desirable feature for program manipulation and verification systems. Prolog and Juniper, which have been implemented with gadgets, support only first-order logic, and the gadget definitions have been concerned only with first-order unification. The key gadget concept, however, of a negotiation process between higher-order functions, appears to extend naturally to the higher-order unification of λ -Prolog, as do the variable abstraction and partial evaluation techniques that are the heart of gadget-based optimisation techniques. We are currently working on capturing Huet's higher-order unification algorithm [2] in a gadget framework. There appears also to be a reasonable amount of concurrent processing that can be identified within the unification process; this may well provide a useful level of parallelism in a shared memory environment. Work in these areas is, of course, still very preliminary.

The short-term goal that we plan to pursue is a gadget-based compiled implementation of λ -Prolog that is robust, efficient, and portable to the research community. The declarative and extremely high-level nature of the language allows the compiler to generate "object code" in any portable language that supports the necessary run-time facilities; we expect that Common Lisp will be the initial candidate. A longer term goal is to exploit the parallelism that can be automatically identified in application programs.

We have established an informal working group of researchers from Penn, Duke, Carnegie Mellon, and Unisys to focus thinking on the desirable directions of development and use of λ -Prolog.

8.1.2. Parallelism in Spoken Language Understanding

The Paoli Research Center has recently defined a Spoken Language Understanding program, combining PRC's expertise in natural language text understanding with speech processing expertise from the speech processing group at MIT. We have demonstrated (Section 5) that the search-dominated parsing process for text understanding supports order-10 concurrency in a realistic architectural framework. Combining this with the ambiguity of the acoustic signal analysis, we expect the size of the search space to increase by several orders of magnitude. In particular, "garden path" phenomena can reasonably be expected to grow, increasing the amount of processing needed to establish that a parse cannot be found in a particular region of the search space.. The amount of concurrency, therefore, should grow dramatically.

The focus of our concurrency research will continue to be or-parallelism, allowing the methodology used in the PUNDIT parser experiments to apply to the larger problem. Without straining our current resources unduly, we can expect to be able to

capture the explosion of syntactic/semantic analyses resulting from the indeterminacy of the input signal. With increased resources we expect to be available in the near future, VCR simulations of full sentence parses will be feasible. This approach will help identify areas for improvement of the speech/grammatical processing and coupling, and to estimate the parallel speed-up that may be obtained on a suitable multiprocessor.

8.2. Tools and Environments

The MASC program has provided us with several useful tools, and demonstrated the need for others. This section describes efforts related to new tools for semantics-based programming debugging and new applications for the MASC simulation technology.

8.2.1. Programming Environment for Implicitly Parallel (Declarative) Languages

The MASC program included the development of application programs in Juniper-B1, which was a challenging but richly rewarding endeavor. While we demonstrated that the language supported the application area reasonably well (several deficiencies were identified; see Appendix B), a major impediment to progress was the difficulty of debugging programs. The Juniper-B1 implementation is based on demand-driven or lazy evaluation. A (non-gadget) logic interpreter implemented as special combinators thus provides an evaluation order that is non-intuitive at best. Our experience shows that it easily becomes counter-intuitive, and presents additional obstacles to a programmer who is already working in an unfamiliar language with novel concepts. Proposed parallel implementations of Juniper would compound the problem by relaxing the ordering constraints of set elements, which is deterministic in the sequential implementation, and evaluating expressions in anticipation of their use, perhaps speculatively. Rather than require the programmer to become familiar with the operational semantics of the implementation, we have designed a debugging system that is based on the declarative semantics of the language and *demand evaluation*. The concepts of the debugging system generalize easily, and a generic debugging framework appears to be feasible that would be usable for a variety of declarative languages and implementations.

Our approach to debugging depends on the declarative semantics of the programming language, hiding the operational semantics of the implementation. It integrates well with current research in program development environments, which use the syntax and, increasingly, the semantics of the language to customise the environment to optimise the development process. The key concepts are *demand evaluation* and *incremental evaluation*, which we use to denote an interactive evaluation process driven by the user. A typical scenario involves the user testing a program unit to determine whether it provides expected results. At the top level, the debugger accepts expressions involving the unit and returns their values. If an anomalous value shows up, the user can step through the top-level elaboration of the expression (replacing a

function application by the function body, for instance), and can specify particular subexpressions to be evaluated. In this way, the user can quickly isolate the source of the anomaly.

We envision a system that is integrated with a modern workstation environment. The user can demand evaluation of an expression simply by identifying the expression (using a syntax-directed editor) and specifying a degree of evaluation; in some cases evaluation to the final value will be desired, while in others it may be more revealing to see the arguments of the top-level functor. In either case, the details of the reduction process are hidden from the user, and only the declarative semantics need be consulted.

By utilizing a standard front end, the user interface can be made standard across a range of language. To make the system more portable, we envision a standard interface to the implementation, which will require only a small layer of software to mediate for existing implementations, and that can be built into new implementations. By using the standard implementation of the language, we avoid having a second "debugging" implementation, which typically is expensive to develop and may contain different implementation errors than the standard implementation.

8.2.2. Ada tasking

Another application of the VCR involves the abstraction of the interactions of the tasks in an Ada task suite. Given the assumption that each task's actions are independent of previous actions (which is called for in the Ada specification), there is limited need to model the state of each task; this makes the VCR ideal for abstracting the relevant behavior and simulating it under different system architecture assumptions. Use of the VCR would allow experimentation and performance prediction for systems with varying numbers of tasks, processors, and messages. Our thinking on this area still very preliminary, but there appears to be sufficient interest in the prediction of Ada performance on highly parallel systems to justify further research into the approach.

8.3. References

- 1 G. Nadathur, A Higher-Order Logic as the Basis for Logic Programming, Ph.D. Dissertation, University of Pennsylvania, May, 1987.
- 2 G. Huet, A Unification Algorithm for Typed λ -Calculus. *Theoretical Computer Science* 1, 1975, pp. 27-57.

Appendix A

Issues in the Definition of Juniper

by

Tom M. Blenko

Table of Contents

Section	Title	Page
A.1	Language definition of Juniper	A-1
A.2	Definition of Juniper-A	A-3
A.2.1	Functional invocation of the logic-programming component	A-3
A.2.2	Sharing of data structures between logical and functional components	A-3
A.2.3	Logical variables returned to the functional component	A-4
A.2.4	Comments on determinism, parallelism, and parallel operations	A-4
A.3	Implementation of Juniper-A	A-11
A.4	Juniper-B languages	A-12
A.4.1	Definition of Juniper-B1	A-12
A.4.2	Implementation of Juniper-B1	A-24
A.5	Issues for further Juniper-B language development	A-25
A.5.1	Control constructs for clause selection	A-25
A.5.2	Other control constructs	A-28
A.5.3	Addition of the <code>call()</code> predicate/operator	A-30
A.5.4	Unreconstructed meta-operators	A-30
A.5.5	<i>assert()</i> and <i>retract()</i> operations	A-31
A.5.6	Explicit logic databases	A-31
A.5.7	Negation by failure	A-31
A.6	Definition of Juniper-C	A-35
A.6.1	Addition of full functions in the logic-programming component	A-35
A.7	Semantics of Juniper	A-37
	References	A-38

Appendices

A-A	Syntax of Juniper-B1	A-41
A-B	Reconstruction of meta-operators	A-43
A-C	Proof-stream interpreter for Juniper-A	A-47
A-D	Success-continuation interpreter for Juniper-A	A-49

Appendix A

Issues in the Definition of Juniper

This section was written by Tom Blenko in late 1986, during the definition of Juniper-B1, and reflects the state of the program's thinking about the language design issues then under consideration. Some of the issues listed were resolved and went into the language definition, and others become moot as the language evolved.

Bill Hopkins edited the section to remove incomplete and irrelevant material.

A.1. Language definition of Juniper

The language definition strategy for Juniper is to construct a sequence of languages that provide successively closer approximations to the eventual goal, a fully-integrated functional/logical programming language. This approach has several advantages: it allows us to incorporate solutions to particular language-definition issues as they become available here at SDC or from other sources; it encourages incremental modification of the implementation strategy as additional constructs are introduced; and it provides a relatively concrete measure of progress in the language development task.

The language definition sequence has been divided into three principal subsequences. The Juniper-A sequence consists of a single, minimal language containing functional and logic-programming components, but providing limited capability for the components to interact with each other. Juniper-A is of little use as an application language, but provides a starting place for addressing the issue of parallelism in Juniper, and for addressing issues related to its implementation.

Languages in the Juniper-B sequence will incrementally introduce constructs of increasing power that permit functional and logic-programming components to invoke each other recursively. In particular, a set-generator expression in the functional component will generate the sets of bindings satisfying a particular goal in the logic-programming component, while logical terms in the logic-programming component will, in certain cases, be treated as executable (reducible) functions. These bridging constructs will implement the smooth transfer of control between functional and logic-programming components that is necessary in order to obtain the desired synergism. They also have sufficient power to permit the introduction, especially into the logic-programming component, of a number of languages constructs generally excluded from the "pure" definitions of functional or logic-programming languages, but which are nevertheless essential to applications programming.

The Juniper-C class will, if pursued, increase the expressiveness of the Juniper-B languages by permitting use of logical variables as arguments to functions. This is currently an open research topic, and the desirability, computability, and implementability of such languages is not yet established. However, it may constitute a powerful

extension to the Juniper-B languages, and a possible path for the development of Juniper.

In pursuing the definition of Juniper, which is the effort catalogued in this paper, we need both to establish that particular language features are practically attainable and to ascertain for ourselves that additional features are not currently practical. We anticipate that definition of some member of the Juniper-B class is our minimal target. To date, we have defined and implemented the initial member of the sequence, Juniper-A, and are engaged in defining and implementing its successor, Juniper-B1. Several potential extensions of Juniper-B1 are examined in a later section of this paper.

A.2. Definition of Juniper-A

Juniper-A is a minimal version of Juniper. It consists, roughly speaking, of the functional language SASL [40] plus pure PROLOG, i.e., PROLOG without the *call()*, *not()*, *cut()*, *assert()* and *retract()* predicates/operations. The syntax of Juniper-A is a subset of the Juniper-B1 syntax (which is included in Appendix A), and is a modest extension of the SASL syntax that is designed to be compatible with the syntax of conventional PROLOG implementations.

A.2.1. Functional invocation of the logic-programming component

The functional component of Juniper-A invokes the logic-programming component using a set generator for variable bindings. That is, if we are interested in the set of bindings for variables ?A and ?B such that the goal $p(?A, f(?B))$ is provable in the logic-programming component, the expression

$$\{ \langle ?A, ?B \rangle \mid p(?A, f(?B)) \}$$

is interpreted in the functional language as a generator returning the set of all pairs of bindings for ?A and ?B such that the goal $p(\dots)$ is satisfied. Note that $f(?B)$ is treated as an irreducible constructor here, as it is in PROLOG, rather than as a reducible function application (as it will be in later versions of Juniper).

In Juniper-A, the clauses from which $p(\dots)$ is to be proven are declared in a fashion similar to the way that SASL function definitions are declared, i.e., as part of a global database defined prior to runtime.

There is no provision in Juniper-A for the logic-programming component to invoke the functional component. In Juniper-B, the logic-programming component will treat terms as functions in the same way that first-order logic does.

A.2.2. Sharing of data structures between logical and functional components

An issue that needs to be addressed immediately is the sharing of data structures between the logic-programming and functional components of Juniper. Logic-programming languages take a very liberal attitude toward constructors — every functor serves as a constructor. It seems likely that we would like to back away from that. With an eye toward definition of functional (reducible) terms in Juniper-B, Juniper-A includes only the list constructor presently in the logic-programming component, and a function-application constructor which replaces the arbitrary term constructors of PROLOG. This aligns terms in the logic-programming component with those in the functional-programming component, i.e., they are all constructed as dotted pairs or function applications; terms previously interpreted as constructors in the logic-programming language (e.g., $f(a, b)$ in the logic-programming formula $p(f(a, b))$) are interpreted as irreducible function applications in Juniper-A. We also need to support two syntactic forms for function applications, the one inherited from the functional

language, (*function arg*), and the one inherited from the logic-programming language, *function(arg)*.

For the implementation of Juniper-A, then, functional-programming and logic-programming lists are assumed to be interchangeable; function applications take the functional-programming (SASL) syntax and are reducible unless they appear in a logic-programming context (e.g., in the definition of clauses in the database).

A.2.3. Logical variables returned to the functional component

Logical variables may appear in the functional programming component since they can occur free in bindings returned from a set-generator expression. In Juniper-A and Juniper-B languages, however, logical variables are not recognized *per se* in the functional component. At present our approach is to treat them as constants in the functional component. This choice will require the implementation to guarantee referential transparency for variables returned from set-generator expressions, e.g., that multiple instances of an expression generate the same variables in the same order, and that equivalence of these variables be decidable in the functional and logic-programming contexts to which they are returned.

A.2.4. Comments on determinism, parallelism, and parallel operations

Juniper differs significantly from the major existing proposals for parallel (concurrent) PROLOG's [9, 35, 41]. The chief reason is that referential transparency, and hence determinism, plays an important role in functional languages, and it appears that determinism in the functional component of Juniper requires determinism in the logic-programming component. We consider this issue and the interplay between determinacy and parallelism in the remainder of this section.

Referential transparency and determinism

An important property of functional languages is *referential transparency*, which requires, among other things, that an expression evaluates to the same value irrespective of its location in the program. Referential transparency is important because it results in reduced program complexity (hence claims of an order-of-magnitude greater code density for functional programs compared to imperative programs) and because it greatly simplifies the parallel implementation of programming languages (hence the choice of functional user languages for dataflow architectures). Preservation of referential transparency is therefore an important constraint on the definition of Juniper.

If we were to include concurrency constructs such as non-deterministic clause selection in the logic-programming component, reduction of the goal

```
?- shuffle([Z]).
```

using the clauses

```
shuffle([a,X]) :- shuffle([X]).  
shuffle([b,X]) :- shuffle([X]).
```

would produce an indeterminant solution. This behavior is in fact permitted in several of the proposed parallel (concurrent) logic-programming languages [9, 35, 41]. We have assumed as a starting point that the functional component of Juniper must be deterministic in order to maintain referential transparency, and it appears that the same must also hold in the logic-programming component. Juniper will therefore be less expressive than the concurrent logic-programming languages.

Sources of parallelism

One of our principal goals is to exploit parallelism in Juniper programs. Several sources of parallelism, known from the study of functional and logic-programming languages, are discussed in greater detail below:

- tasks or processes (for concurrent languages)
- logical OR-parallelism
- logical OR-parallelism for concurrent PROLOGs (PARLOG, CP, GHC)
- logical AND-parallelism
- functional normal-order evaluation with strictness
- functional applicative-order evaluation
- parallel operations on data structures
- anticipatory evaluation

Our assumption of referential transparency excludes indeterminate, user-level concurrency constructs, and for the time being we are excluding explicit concurrency constructs altogether. A consequence is that a major source of explicit parallelism at the user level (tasks and processes) is disqualified altogether. It is especially important, therefore, to examine the remaining sources of (implicit) parallelism that we hope to exploit in the implementation:

- (1) *logical OR-parallelism* — A logical OR-parallel computation is an n -way branch, with each branch representing the independent reduction of a common goal by a distinct logical clause. Each branch produces a resultant set of bindings (or failure), and the result of the computation is a bag containing all of these sets.
- (2) *logical AND-parallelism* — A logical AND-parallel computation produces a bag of variable-binding sets. Each variable-binding set produced represents the reduction of a set of sub-goals (which taken together constitute a conjunctive goal), in

which each sub-goal is reduced in the environment (binding set) produced by its predecessor in order to generate an augmented environment for use by its successor. The computation of a binding set has a strongly serial character, and relaxation of this serial character (exploiting AND-parallelism) is a goal of current research efforts. An AND-parallel computation may produce more than one binding set because reduction of one of its sub-goals using OR-parallelism may result in more than one possible augmentation of the environment received from its predecessor.

(3) *functional normal-order evaluation with strictness* — Normal-order evaluation is a strictly sequential redex selection rule (leftmost redex is always reduced next), and has the desirable quality of preserving the Church-Rosser property (that the computation will terminate with the normal form of the expression if there exists any selection rule under which it terminates). One way of relaxing the selection rule in order to obtain more parallelism, while preserving the Church-Rosser property, is to allocate computational resources in such a way that non-leftmost redexes are reduced concurrently, but *reduction of the leftmost redex is never indefinitely delayed*.

A selection rule that has been included in existing functional languages is *strict evaluation*, a mechanism by which it may be specified, for a particular function, that one or more of its arguments must be reduced before the function is itself reduced. *if-then-else()*, for example, can be viewed as a higher-order function that is strict in its first argument (the condition). If a function is strict in two or more arguments, those arguments can be evaluated in parallel, so that strictness can provide an additional source of parallelism. Strict evaluation does *not* necessarily preserve the Church-Rosser property since strict evaluation of an argument may indefinitely delay leftmost reduction.

(4) *functional applicative-order evaluation* — applicative-order evaluation is often identified with call-by-value procedure invocation. Since functional expressions are referentially transparent, all arguments to a function can be evaluated in parallel, and this is a possible source of significant parallelism. Unfortunately, applicative-order evaluation does not preserve the Church-Rosser property, since the evaluation of an argument may produce a non-terminating computation, even though it may be the case that the function evaluation could have proceeded to completion while ignoring the value of that argument.

(5) *parallel operations on data structures* — a representative example of this form of parallelism is parallel addition of two vectors. This is a common and well-studied source of parallelism. Unfortunately, there is little support for data structures in functional or logic-programming languages *per se*. The absence of data and control structures especially suited for these operations makes the general problem of extracting the parallelism much more difficult. In many languages, the operations

are effected by loops; in functional and logic-programming languages they are most commonly defined using recursion, which in practice tends to be quite costly.

(6) *anticipatory evaluation* — Anticipatory evaluation is a special type of eager evaluation. It is a less well-studied source of parallelism than those listed above, and it may prove useful for our implementation. Anticipatory evaluation performs computations that *might* be needed for the execution of a program. For example, given a run-time conditional

IF <condition> THEN <then-computation> ELSE <else-computation>

and excess computing capacity, it may be the case (in a referentially-transparent language) that evaluation of <then-computation> and <else-computation> may begin prior to termination of <condition> as long as an idle processor is available. If the processor is allocated fairly between the two, then irrespective of the result of the <condition> computation, evaluation of the expression as a whole will be advanced by the time obtained from the idle processor with a 50% effective utilization factor.

It is, of course, the case that the effective utilization factor drops as we anticipatorily evaluate more deeply-nested conditionals. Inside three nested conditionals, for example, the effective utilization is about 10% since there are eight candidate continuations from the top-most condition. But consider a rule of thumb for multi-processor systems, that n processors produce $\text{square-root}(n)$ speedup: $\text{square-root}(n)$ speedup implies roughly $1/\text{square-root}(n)$ processor utilization. On a 16-processor system, suppose 25% of the cycles are applied to the critical-path computation and 25% of the cycles are charged to multi-processor management overhead for the critical-path computation: then we are left with 50% of the cycles unused. If we allocate equal time-slices to each of our eight three-deep nested conditionals, each effectively gets a full processor without affecting the progress of the critical-path computation, and yet, since one of the eight conditional outcomes will eventually be selected (and will have been partially computed), we are effectively doubling the performance of the system as a whole. Admittedly this is a rough example.

An important application of anticipatory evaluation in functional and logic-programming languages is recursive procedure or loop evaluation. If potential branches in the computation have resources allocated in proportion to their depth of nesting within run-time conditionals (which would, for example, be equivalent to relative depth in the recursion), we immediately have a prescription for parallel computation of recursive or iterated procedures. The current level receives all the resources it can use, the next, which may never be executed, receives half, and so forth, so that the priority of all anticipated branches decreases with distance down the recursion tree. This application of an anticipatory evaluation strategy

provides a simple, general approach to parallelizing the recursive procedure/loop while accommodating the decreased likelihood of the anticipated computations ever taking place.

Note also that anticipatory evaluation partially addresses one well-known objection to parallel computing: that some problems/computations are serial by nature, so there is little that can be done to speed them up. What we are doing by performing anticipatory evaluation, interestingly enough, is non-deterministically simulating a deterministic computation by guessing which branches will be taken and using sufficient parallelism to ensure that the non-deterministic computation proceeds more quickly than the deterministic one.

Our next task, then, is to map these sources of parallelism onto the parallel processing capacity available.

Implicitly parallel computations

We have identified three classes of implicitly parallel computations among the sources listed above. Their identification is helpful both in defining the programmer's view of the language semantics and in specifying the mechanisms required for the implementation of the language.

1. computations returning a single value

Examples: conjunctions and disjunctions

2. computations returning a structure containing independently-computed values

Example: functional map operator — `map(func,list)`

3. computations returning bags (unordered collections) of independently-computed values

Example: $\{ ?A, ?B \mid p(?A, f(?B)) \}$

Class 1 is important to our implementation, as will be discussed below. Class 2 is important because it represents (among other things) parallel operations on data structures. We expect class 3 to be unimportant to us because, under the restriction of referential transparency, it is properly included in class 2.

As an example of class 1 implicit parallelism, we propose the following semantic definition for parallel evaluation of conjunctions (termed "finite failure" [33]).

Evaluation of a parallel AND expression yields *FALSE* if and only if evaluation of one of the conjuncts returns *FALSE*. If *all* conjuncts return *TRUE*, the expression returns *TRUE*. Else the computation fails to terminate.

A similar definition can be made for parallel OR computation:

Evaluation of a parallel OR expression yields *TRUE* if and only if evaluation of one of the disjuncts returns *TRUE*. If *all* conjuncts return *FALSE*, the expression

returns *FALSE*. Else the computation fails to terminate.

In the implementation, fair allocation of cycles among the individual conjuncts does not generally provide an optimal solution (in fact it is likely to be non-optimal), although it does assure termination whenever possible (i.e., it is complete with respect to the desired semantics). In the case of parallel AND computation, the optimal solution consists of "guessing" and evaluating the conjunct which will fail, because the optimal ordering in the case that *FALSE* is returned consists of computing the failing conjunct first, and the length of the computation is independent of the ordering if *TRUE* is returned, or if the computation fails to terminate.

The chief feature of class 1 computations, however, is that when the result becomes available, the program performs a non-local exit [3, 21] (from the failing conjunct, in this example), and all other conjunct evaluations underway are suspended indefinitely. For our purposes, an especially important representative of computations in class 1 is unification.

The computations in class 2 are fairly conventional. Lack of support in functional/logic-programming languages for parallel access to data structures means that the primary access method is recursive or iterative search/enumeration of a structure, so class 2 computations are important both for characterization of functional and logic-programming operations, and as a source of implicit parallelism to be exploited.

Class 3 contains an important construct: the set generator (really a bag generator) that provides the interface between the functional-programming and logic-programming components of Juniper. We have an unresolved question, however, as to whether class 3 can, for our purposes, be properly included in class 2.

Our constraint of referential transparency requires that any operation on the set returned from the set-generator be deterministic, so that non-deterministic operations such as *anyof(Set)* are forbidden. For practical reasons (e.g., debugging), it need not only return the same value no matter where in the computation it appears (the referential transparency constraint), but it needs to return the same value on each invocation of the program (which would not otherwise be required).

Suppose the operation we wish to perform is set enumeration. Then we have a number of possibilities for fixing the ordering:

- 1) Elements of the set are returned in the order in which they become available at runtime (within a parallel processing environment). This leaves the ordering up to the implementation. If we can assure ourselves that the implementation will return the same ordering on each invocation, and if we are willing to weaken the requirement that the value be unchanged for repeated invocations only in the case that the underlying system is unchanged, and if we are willing to permit different valuations for an expression occurring in different (and otherwise unrelated) programs, then this may be a satisfactory approach.

2) Elements of the set are returned in an order defined by the compiler, independent of the system configuration, and unknown to the programmer in advance. If we allow the compiler to choose an ordering arbitrarily, the problem we encounter is that the arbitrary ordering of an infinite list may be an inconvenient one. For example, the naive PROLOG program

```
ancestor(X,Z) :- ancestor(X,Y), ancestor(Y,Z).  
ancestor(John, Martha).
```

does not include $?X = Martha$ in its (least fixed-point) solution to the query $ancestor(John, ?X)$, which is probably not the programmer's intention. Since the compiler cannot know which elements take longer to compute, the programmer is left very much at the mercy of the compiler's arbitrary behavior.

3) Define a convention for set ordering that is known to the programmer, implemented by the compiler, and guaranteed by the system implementation. This is the most familiar approach, and one we are assured the programmer can make use of. Note that one consequence is to restrict all class 3 computations to be class 2 computations. Unfortunately, it also forces serialisation on the computation, which reduces our ability to exploit parallelism, as well as complicating the programmer's model (relative to 1) above).

Determinism and expressiveness

As mentioned earlier, an immediate consequence of our insistence on determinism in Juniper is loss of the elegant expressiveness available in the proposed concurrent logic-programming languages, PARLOG, CONCURRENT PROLOG, and GHC. There are, however, significant benefits that should not be overlooked:

1. Juniper has a significantly simpler semantic interpretation than concurrent languages, and one which is much closer to those of conventional, non-concurrent programming languages. This has implications as far as ease of programming, training of programmers, and the overall cost of developing and debugging applications.
2. It is not known, in general, how to compile concurrent languages, and this has been a major obstacle for implementors of PARLOG [9] and CONCURRENT PROLOG [25, 41]. As a result they have had to seriously weaken the expressiveness of the languages (resulting in the elimination of non-deterministic clause-selection, for example).

A.3. Implementation of Juniper-A

The initial implementation of Juniper-A consisted of Turner's SASL interpreter, with the front-end modified to recognize the syntax of logic programs, together with a pure PROLOG interpreter written in SASL. Only one small change to the SASL interpreter's intermediate-level language (combinator expressions) and to the combinator interpreter was required.

The initial implementation was problematic, however, because it required the SASL component of the language, in which the logic-programming interpreter is implemented, to explicitly recognize unreduced function applications. That is to say, a predicate for recognising function applications, as well as *function()* and *args()* selector functions, had to be added. The additions are not consistent with the semantics of SASL (or other functional languages). The logic-programming interpreter will be reimplemented, therefore, at the combinator-interpreter level (for Juniper-B1) with substantial benefits in terms of improved performance.

There remains, however, a related problem because SASL uses lazy evaluation. If an expression *e1* is defined as *(hd e2)*, the internal representation of *e1* is set to a function application node with function *hd* and argument *e2*, rather than to the (reduced) value of *(hd e2)*. The consequence is that if the value of *(hd e2)* is, for example, *(f x)*, and the logic-programming component attempts to unify a function application with *e1*, it will find itself trying to unify the function slot with *hd* rather than with *f*. Our solution has been to mark these lazy constructions at the time they are created so that they may be partially reduced (in this case to *(f x)*) at unification time.

Implementations of interpreters for PROLOG and logic-programming languages using a functional language have been discussed by Carlsson [4] and Nilsson [27]. The two principal approaches are *success-continuation* and *proof-stream* interpreters. Our combinator-level implementation will follow the form of a proof-stream interpreter, which, unlike the success-continuation interpreter, does not require higher-order functionality.

Additional information about the implementation is available in a companion document, *Juniper language definition (working paper)*.

The code for SASL-language proof-stream and success-continuation interpreters for Juniper-A is included in Appendices C and D.

A.4. Juniper-B languages

Juniper-B is a class of languages that, unlike Juniper-A, include reducible, grounded (logical-variable free) functional terms in the logic-programming component. This, along with the set-generator capability from Juniper-A, permits the logic-programming component to invoke itself recursively, and, in turn, allows the definition of powerful logic-programming control constructs. Introduction of this capability also requires the definition of control policies governing the unification and reduction of terms. Altogether, the Juniper-B languages should prove useful for application development (as Juniper-A is not because of the absence of sufficient constructs). The development of Juniper-B languages will consist of several definition/implementation cycles, Juniper-B1, Juniper-B2, etc., in order to incrementally incorporate and evaluate these and other features.

A.4.1. Definition of Juniper-B1

Juniper-B1 includes the following capabilities not present in Juniper-A:

- 1) The ability to make calls on functions by referencing them as terms in the logic-programming component. The current plan is to restrict these function calls to first-order functions, in order to avoid higher-order unification, although it is not clear how this restriction will be enforced.
- 2) With the addition of reducible terms to the logic-programming component, some policy is necessary for deciding when a functional term should be unified and when it should be reduced. This is equivalent to augmenting the unification algorithm with equality-testing modulo reduction. We require a policy that preserves expected termination conditions, conforms to a semantic model the programmer can employ correctly and reliably, and is simple enough computationally to permit realistic execution times for user applications.
- 3) Implementation of a PROLOG-style *call()* operator, and, as an immediate consequence, a *fail()* operator serving in the same role as the PROLOG *not()* operator. *call()* is implemented using the capability listed in 1) above to recursively invoke a new instance of the logic-programming interpreter.
- 4) Using *call()* and *fail()*, an *if-then-else()* construct is defined that is more expressive than the infamous PROLOG *cut()*. Using *call()* and *if-then-else()*, several of the meta-operations present in PROLOG are reconstructed, as detailed in appendix B. We expect these reconstructed operators to constitute a "critical mass" for Juniper-B1 to be used as an implementation language in the development of serious applications.

Syntax

The syntax for Juniper-B1 is included in Appendix A. It is chosen to be as compatible as possible with the SASL syntax and with CPROLOG and DEC-10 PROLOG syntax.

Function calls from the logic-programming component

The addition of functions to the logic-programming component of Juniper-A requires the solution of two sub-problems: first, the addition of *pure* functions, i.e., functional terms which do not contain ungrounded logic-programming variables at reduction time, and second, the addition of *full* functions, i.e., pure functions whose domains have been extended to include logical variables.

Juniper-B1 permits only pure functions, so functional terms containing ungrounded logical variables will cause both reduction and unification to fail.

A major consequence of including both bridging constructs is that it permits each component (functional and logic-programming) to invoke itself recursively. Since the functional component is higher-order, this does not contribute to its expressive power; the logic-programming component, on the other hand, benefits substantially from the *call()* and *fail()* operators that are introduced, as will be shown below.

Control over unification and reduction

In order to add functions to the logic-programming component, the issue of control over functional reduction in the logic-programming context must be addressed, preferably in such a fashion that the Church-Rosser property is maintained.

The problem

Under its usual interpretation, unification requires strict reduction of each functional term. This may yield a non-terminating computation, even in the case that a different, arguably correct, ordering of the reductions would terminate. A simple example of non-termination is taken directly from the study of functional programming: if we wish to unify the formulae

`p([?First;?Rest])`

and

`p(primes())`

where *primes()* is a generator of the prime numbers, strict evaluation of *primes()* would require a non-terminating computation (the generation of the whole list of prime

numbers). One solution to this, based on the lambda calculus and previously applied to functional programming, is lazy, or demand-driven, reduction of terms. In this case, lazy reduction would reduce the argument to $p()$ until it consisted of a list constructor, bind $?First$ to a not-necessarily-reduced functional expression representing the head of the list, and bind $?Rest$ to a not-necessarily-reduced functional expression representing the tail of the list. Subsequent operations on $?First$ and $?Rest$ may force their further reduction.

More difficult are control issues which arise due to combining control over functional reduction and control over logic-programming resolution. Using the model of unification assumed in Robinson's reduction rule for unification [34] discussed below in the section Reduction Rules, the Boolean value of the unification of two terms is the conjunction of Boolean values obtained by recursively unifying each pair of corresponding sub-terms. The issue of control over the unification, then, reduces to control over computation of a conjunction. If, for example, we wish to unify

$p(1,0)$

with

$p(f(0),1)$

we need to evaluate the conjunction

$1=f(0) \ \& \ 0=1$

Then $f(n)$ evaluates to a series that approximates the square root of n , and the series converges only on the open interval $(0, \infty)$, there may be a problem with termination of $f(0)$, so that evaluation of the conjunction terminates only if the second conjunct is evaluated (to produce *FALSE*) prior to the first. Since it is "obvious" to the programmer that the unification should fail, we would like to guarantee that such is the case in Juniper. Completeness and termination of conjunction evaluation is discussed above in the section Implicitly parallel computations.

More difficult and unsolved problems exist. One can imagine, and we need to examine, cases in which terms may be unified without reduction, or with limited reduction, or both. Lynette's example

$+(?X,3)=+(2,?Y), ?Y=5$

clearly requires a more sophisticated model of the interplay between unification and reduction. [16, 19, 38, 39] contain proposals for incomplete control strategies.

We also need to consider how Skolem functions may be correctly handled (and possibly other functions that are undefined).

At present, we know that the following views of the problem, at least, must be accommodated in any solution:

1. The restriction to pure functions simplifies the unification that needs to be done. In particular, the only term containing a logical variable that can be passed to the unifier is a term which is a logical variable. All binding is uni-directional – from a term to an unembedded variable, and all values of bound variables are either grounded or variable equivalences.
2. Unification algorithms have been well studied [11, 24, 30, 36], and since we are trying to augment unification of logical terms with reduction of functional terms, Kahn [16, 17], among others, has suggested that adding functions to a logic-programming language is equivalent to augmenting the unifier's equality-reasoning capability. The augmented procedure is termed ϵ -unification [ref. ?]. We would like some notion of how conventional unification algorithms can be extended to include equivalence of functional and normal forms.
3. A sufficient approach to the problem consists in defining an equality relation that, subject to issues of effective computability and control, subsumes the equality relations associated with functional and logic-programming languages.

In particular, functional-language equality is based upon equality of normal forms, i.e., two terms are equal if and only if their normal forms are identical. The definition is actually weaker than that because languages like SASL do not allow comparison up to normal forms of terms used as functions. It is weaker still than it otherwise might appear because the set of normal forms for functional languages (and moreover for combinator expressions) is larger than the set of equivalent lambda expressions due to uncurrying. That is, if we translate normal-form combinator expressions to the corresponding normal forms in the lambda calculus, we find that in some cases more than one combinator expression maps to the same lambda expression – so that equality of lambda expressions is stronger than equality of combinator expressions or functional-language expressions.

Equality in logic-programming languages is equality in the sense of unifiability, and this incorporates a strong element of syntactic equivalence. It has no sense of equivalence through reduction.

What is needed, then, is a way of defining equality so that the equality relation is, in sensible cases, effectively computable, and so that both kinds of equality, equality through reduction to normal (or intermediate) forms, and the strongly-syntactic style of equality, are subsumed.

A unification/reduction policy for Juniper-B1

The unification/reduction policy for Juniper-B1 is chosen to be as simple as possible. This satisfies several goals simultaneously:

1. It maintains a substantial degree of backward-compatibility with both logic-programming and functional programs. This permits the strengths of both programming paradigms to be preserved.
2. It presents the programmer with a simple model of what to expect. That is to say, the operational semantics is straightforward and accessible to the programmer.
3. It preserves the Church-Rosser property.
4. It maintains some ambivalence toward logic-programming constructors, and towards functions, such as Skolem functions, that are undefined.

The policy is as follows:

1. All reduction is demand-driven. We will not guarantee that non-demanded expressions will remain unreduced, but only that non-terminating computations will be undertaken only in those cases where they are strictly required.
2. The normal operation of the unifier will be to reduce all functional expressions required for unification to normal form. This policy is non-optimal in that it forces reductions that might not otherwise be required. For example, if equality of two terms, t_1 and t_2 , is being tested, and it happens that $t_1 = t_2 = f(z)$, then it would be possible to verify the equality by recognizing that in both terms the functions f are the same and the arguments z are the same, so they must therefore evaluate to the same value (even though we don't bother to determine what that value is).

In a straightforward implementation of a functional language, once the term t_1 has been reduced, even if the reduction takes place, for example, in the context of a failed test for equality, the original form is lost. Subsequent tests for equality with equivalent terms will then require reduction of all terms to normal form, so that $f(z)$ may be reduced to normal form multiple times in order to satisfy multiple tests for equality with t_1 .

For Juniper-B1, we have chosen to *always* force the reduction with the ideas that

- 1) the performance penalty is probably not too great relative to the

less eager approach in an implementation as outlined above;

2) the chief benefit is in providing the programmer with a simple model of what is to happen operationally, since the resulting reduction rule is quite simple;

3) at this point performance is not a critical issue, and the issues of alternate approaches to the implementation and/or more complex reduction rules can be left until we have more experience in this area.

3. Undefined functions will motivate the definition of a new kind of normal form. That is, in cases where a function is not defined, equality testing via the method outlined in 2. above will be applied. For example, the terms

$p(?X, f(2))$

and

$p(1, f(2))$

will unify successfully even when the function f is undefined.

The consequences of this portion of the policy are two-fold. First, it allows Skolem functions to be treated in a very natural fashion, which is highly desirable in consideration of logic-programming's origins in first-order predicate calculus and the fact that the definitions of Skolem functions are not generally known. PROLOG makes allowance for this, or course, by treating the functions as constructors (and hence effectively irreducible). The second consequence is that, in spite of our earlier admonition about restricting constructors and data structures to be identical with the functional-programming forms, we are providing a means here to do precisely the same thing under more restrictive conditions, i.e., the requirement that the constructor name have no associated functional definition, and the policy of reducing embedded terms when permitted under this policy.

Implementation of *call()* and *fail()*

The mechanism of Juniper's logic-programming *call()* operation is for the logic-programming component to invoke the functional-language component, which in turn invokes a distinct instance of the logic-programming component. The equivalent of a PROLOG clause

`:- call(goal(?A,f(?B))), q(?B).`

might look something like

**`p(goal(?A,f(?B)))
:- non_empty({ <?A,?B> | goal(?A,f(?B)) }), q(?B).`**

in Juniper. The **`{ <?A,?B> | goal(?A,f(?B)) }`** construct is a functional expression, a set generator which reinvokes the logic-programming component to produce a set of variables bindings. **`non_empty()`** tests for the existence of a set of bindings for **`?A`** and **`?B`**.

Since Juniper-B languages are restricted to pure functions, we need to take a closer look at how the set-generator bridging construct is defined in Juniper-B1, and in particular how **`call()`** can be implemented using it. Clearly, the set generator must contain logical-variables, but note that referential transparency is preserved:

1) All logical variables in the goal part of the expression are either fully grounded, or present in the template part and treated as unbound. The effect is to prohibit (in the sense of pure functions) "free" variables from appearing in the set-generator expression. All input values are present in the form of grounded variable bindings at the initiation of expression evaluation. Variables introduced in the template (which is a tuple of distinct variable names) close the scope of equivalent names in the goal, and are consequently unaffected by variables of the same name occurring in the calling context.

2) Since variables appearing in the template are place-holders, i.e., their names are irrelevant except for identifying a slot (or slots) in the goal whose value(s) should be simultaneously bound, there is no sense in which the names of variables in the template ever need to be thought of as corresponding to, or failing to correspond to, logical variables in the context from which the expression is called. Result values are available only via the sequence of binding sets returned as the value of the expression. Consequently, their binding to variables in the context must be performed explicitly, and such binding always results in elimination of variables named in the template. Note that the equivalencing of template variables effected by the goal-reduction operation is recaptured in the logic-programming context by unifying logical variables with the bindings returned.

[DB variables]

We propose, then, the following code for definition of a **`call()`** operator in the logic-programming component of Juniper-B1. **`extract_vars()`** depends on a **`var()`**

predicate that we have not introduced, but which is defined using the results of the **Reconstruction of control and meta-operators** section below, as is *if-then-else*(*P,Q,R*) (syntactically $P \rightarrow Q ; R$). Note that, while we define *var()* later in terms of *call()*, a considerably less general use of the bridging constructs can be used to define *var()*, so the definition below is not circular.

```
call(?P) :-
    unify(extract_vars(?P),?Template),
    call1({ ?Template | ?P },?Template).
```

```
call1([?Soln;?Rest],?Template) :-
    (unify(?Template,?Soln)) -> True ;
    call1(?Rest,?Template).
```

fail() is implemented in a similar fashion. When successful, the set generator returns a sequence of sets of variable bindings. In the case that no bindings are created (when, for example, a grounded term is proven as a goal), the generator returns a sequence of length one, consisting of a single, empty binding set. In the case that the goal cannot be proven, the set generator returns an empty sequence, one containing no variable-binding sets.

```
fail(?P) :-
    unify(extract_vars(?P),?Template),
    fail1({ ?Template | ?P },?Template).
```

```
fail1([],?Template).
```

with *fail1()* checking for the empty solution set rather than attempting to capture the resultant bindings.

PROLOG *not()* computes (NOT EXISTS x $P(x)$), which is logically-equivalent to (FORALL x NOT $P(x)$), rather than the logically-motivated negation operation, (EXISTS x NOT $P(x)$), as discussed in the section **Negation by failure**. Therefore we have chosen to rename *not()* as *fail()*. While the renaming is not compatible with existing PROLOG implementations, it does allow the user to define *not()* selectively in order to force completeness on individual predicate definitions:

```
p(<args1>) :- <body1>.
.
.
.
p(<argsN>) :- <bodyN>.
not(p(?Anything_else)) :- fail(p(?Anything_else)).
```

which recaptures the usual behavior of PROLOG with respect to $p()$, where $not()$ is defined as negation-by-failure.

Reconstruction of control and meta-operators

Many of the meta-predicates (*isvar()*, etc.) in PROLOG can be reconstructed using *call()*, *fail()*, and *cut()*, as we document systematically in Appendix B. Inclusion of these constructs in Juniper-B1 follows from the re-introduction into the logic-programming component of *call()* and a substitute for *cut()*. We demonstrate below how a substitute for *cut()* can be defined using *call()* and *fail()*, which we have already introduced.

Reconstruction of control operators

Using *call()* and *fail()*, it is possible to construct a *cut()* operator, and a more powerful variant of *cut()*, the *if-then-else()* operator. We summarize the construction of *if-then-else()* here; the reasoning is justified in greater detail in the succeeding sections.

1. *cut()* is (almost) equivalent to *if-then-else()*.
2. *fail()* can be defined using *cut()* and *call()*, and, therefore, using *if-then-else()* and *call()*. We propose that, since the operational definition of *call()* and *fail()* is straightforward in Juniper, we should instead define *if-then-else()* using *call()* and *fail()*.
3. Since *call()* and *fail()* suffice to define *if-then-else()*, they suffice (up to syntactic sugar) to define a sequential version of the *select_one* construct described in the Control constructs for clause selection section below.

cut() is (almost) equivalent to *if-then-else()*

A typical use of cut is in a sequence of clauses

```
p(?arg11,?arg12) :- <cond1>, !, <action1>.
p(?arg21,?arg22) :- <cond2>, !, <action2>.
.
.
.
p(?argN1,?argN2) :- <actionN>.
```

The scope of the *cut()* is the entire set of clauses, which are understood to be searched sequentially. This can be equivalently expressed in terms of an if-then-else construct, conventionally ($P \rightarrow Q$);R, in the following way:

```

p(?A,?B) :-  (?A=?arg11, ?B=?arg12, <cond1>) -> <action1> ;
              (?A=?arg21, ?B=?arg22, <cond2>) -> <action2> ;
              .
              .
              <actionN>.

```

where $\text{--}()$ designates unification, and is readily defined by the clause

```

 $\text{--}(\text{?X},\text{?X}).$ 

```

fail() defined in terms of *if-then-else()*, and vice-versa

fail() is readily defined using *cut()*:

```

fail(P) :- call(P), !, fail.
fail(P).

```

or *if-then-else()*

```

fail(P) :- call(P) -> fail ;
          true.

```

However, since *fail()* is an operation that arises naturally in Juniper, we are proposing that *cut()*, or rather its substitute, *if-then-else()*, be defined in terms of *fail()*, rather than the other way around:

```

if-then-else(P,Q,R) :- call(P), call(Q).
if-then-else(P,Q,R) :- fail(P), call(R).

```

Note that (1) if *if-then-else()* terminates, then exactly one of *call(Q)* and *call(R)* has been attempted, and (2) *fail()* produces no bindings, which is exactly the behavior we expected of the original $(P \text{--} Q);R$ construct.

call() and *fail()* suffice to define sequential *select_one*

What we have shown is that *fail()* is sufficient to reconstruct *cut()*, modulo syntactic sugar (in the form of *if-then-else()*). In conjunction with a scoping construct, *cut()* would allow us to define a sequential version of the *select_one* construct described in the Control constructs for clause selection section below, which has greater expressive power than PROLOG's *cut()*.

In PROLOG, the *cut()* operation in the body of a clause for a functor *p()* is understood to have a scope which encompasses the set of all clauses defining *p()* and containing *cut()* in the database. The same effect can be achieved by gathering all the clauses for *p()* that contain *cut()* into an *if-then-else()* construct in the body of a single clause for *p()*. For additional expressive power, we could refine PROLOG *cut()* by adding a scoping construct that defined clause subsets whose *cut()* operators are understood to interact. This would achieve the same effect as writing multiple clauses, each of whose bodies is an *if-then-else()*. We conclude from this that *if-then-else()* is more expressive than *cut()*.

The latter effect could also be achieved more awkwardly in PROLOG itself by using additional functor names to effect multiple scopes.

For Juniper-B1 we have chosen to do without *cut()* and the scoping construct, and to require the programmer to use the equivalent *if-then-else()* forms. Note that a consequence of our choice of *if-then-else()* is to require the programmer to explicitly unify actual arguments with formal arguments.

Example:

```

/*
 * p() is satisfied by bindings from either of two sources. Each source is
 * expressed by an if-then-else condition.
 */

/*
 * Juniper-B1 code for p() using if-then-else()
 */

/* first source has three possible cases */
p(<args>) :- (unify(<args>,<select1.1>), <guard1.1>) -> <body1.1>;
              (unify(<args>,<select1.2>), <guard1.2>) -> <body1.2>;
              unify(<args>,<select1.3>) -> <body1.3>.

/* second source has two possible cases */
p(<args>) :- (unify(<args>,<select2.1>), <guard2.1>) -> <body2.1>;
              unify(<args>,<select2.2>) -> <body2.2>.

/*
 * PROLOG code for p() using cut() and a scoping construct
 */

/* first source has three possible cases */

```

```

cut_scope {
    p(<select1.1>) :- <guard1.1>, !, <body1.1>.
    p(<select1.2>) :- <guard1.2>, !, <body1.2>.
    p(<select1.3>) :- <body1.3>.
}

/* second source has two possible cases */
cut_scope {
    p(<select2.1>) :- <guard2.1>, !, <body1.1>.
    p(<select2.2>) :- <body2.2>.
}

/*
 * PROLOG code for p() using cut() and functor name discrimination
 */

p(<args>) :- p1(<args>).
p(<args>) :- p2(<args>).

/* first source has three possible cases */
p1(<select1.1>) :- <guard1.1>, !, <body1.1>.
p1(<select1.2>) :- <guard1.2>, !, <body1.2>.
p1(<select1.3>) :- <body1.3>.

/* second source has two possible cases */
p2(<select2.1>) :- <guard2.1>, !, <body1.1>.
p2(<select2.2>) :- <body2.2>.

```

Reconstruction of meta-operators

Much has been made of the "non-logical" character of several useful PROLOG operators. We have already discussed the *cut()* operator. In addition, we show in Appendix B that *call()* and *cut()* are sufficient to generate many of the meta-operators in PROLOG. Since *call()* and *fail()* suffice to define *cut()*, they suffice to reintroduce many of the PROLOG meta-operators into Juniper-B1. Our present hypothesis, therefore, is that *call()* and *fail()* (together with a policy governing the interaction of unification and reduction, and a set constructor for calling the logic-programming component of Juniper) suffice to make Juniper-B1 a practical programming language.

As a parenthetical note, we mention here what remains to be worked out in the section Semantics of Juniper below: with regard to the "non-logical" or "meta" constructs, we believe that it is possible to provide a logical semantics, e.g., a first-order predicate logic, which encompasses the logic-programming language component of Juniper including reconstructed "non-logical" operators.

Meta-operators which are not reconstructible using *call()* and *fail()* are discussed below in the section *Unreconstructed meta-operators*.

The only additional operator we wish to introduce at this point, *unify()*, is discussed in the next section. Note that it is required for effective use of *if-then-else()*.

The explicit unify() operator

The power implicit in the variable-binding mechanism of logic-programming procedure calls is demonstrated by the *=()* or *unify()* predicate, which can be defined without *call()* by enlisting the implicit unification (meta-)operation performed in matching goals to heads of clauses:

=(?X,?X).

or

unify(?X,?X).

The test for implicit usage of this mechanism is to force the clause head to be written using a disjoint set of variables as formal arguments. For example, reduction by the clause

p(?X,f(?Y),?X,?Y) :-
 <body>

is the same as reduction by the following clause, which uses *=()* explicitly:

p(?A,?B,?C,?D) :-
 ?A = ?C, ?B = f(?D), <body with appropriate substitutions>.

A.4.2. Implementation of Juniper-B1

The Juniper-B1 implementation is an extension of the Juniper-A implementation.

A.5. Issues for further Juniper-B language development

Juniper-B1 can be extended in several more-or-less independent directions. The language definition process is such that solutions to specific cases of some of the issues raised here have been introduced earlier in the paper. The discussion here, however, reflects our broadest ambitions for what should ultimately be included in the Juniper-B languages.

A.5.1. Control constructs for clause selection

[general issue -- applies also to function defs.]
[also relaxation of serial functional definitions]

OR-parallel selection of clauses

I think we are agreed on the following with respect to OR-parallel control constructs:

- (a) The programmer needs an IF-THEN-ELSE construct.
- (b) OR-parallel clause-selection constructs in Juniper should be deterministic in order to assure referentially-transparent functional expressions. See discussion above in Comments on determinism...
- (c) We can imagine a deterministic guard construct in which at most one guard among a specified set of clauses may succeed. Definition of the deterministic guard has significant implications for compilation and execution of the language.
- (d) A smart compiler can translate an instance of (a) to a special case of (c) which does not recompute the failure of a preceding condition (guard) in the IF-THEN-ELSE.
- (e) [Non-exclusive OR parallelism]

This leaves us with three major issues to consider:

- (i) How is disjointness of guards in (c) to be assured? A set of syntactic restriction rules would be desirable, but may prove difficult to construct since the language and the syntax has been defined with other purposes in mind. Semantic restrictions may place a more severe burden on the programmer, and checking for semantic disjointness in the compiler may require detection of undecidable or uncomputable conditions.
- (ii) Are there other restrictions which should be placed upon the computation

permitted within guards? It seems likely that we want to allow at least the equivalent of the computation required for the pattern-directed matching of heads of clauses in PROLOG, which consists of explicit calls to the unifier. In addition to constraints on the syntax or semantics of guards for the purposes of addressing issue (i), implementation considerations may dictate further restrictions, e.g., prohibition of universal variables in guards [9, 41] or restrictions on the cost of computations performed in guards [29]. At this time, we don't have a very good idea what the issues or options are with respect to imposing these restrictions.

(iii) How desirable is it to have the full, which is to say non-serial, guarded clause selection construct? Would a serial construct (Juniper-B1's *if-then-else()*, for example) suffice? And would the non-serial construct put an unusual or unwanted burden on the programmer, who almost certainly has experience with serial clause definitions, or function definitions, or case constructs?

Syntax of guards

For the purposes of discussion, I am proposing the following syntax for a guarded, deterministic, OR-parallel clause-selection construct:

```

<logic_clauses> ::=
    <logic_clause>
    | <logic_clauses> <logic_clause>
<logic_clause> ::=
    <simple_logic_clause>
    | <guarded_logic_clause>
    | <select_one>
<simple_logic_clause> ::=
    <goal> :- <goal_list> .
    | <goal> .
<select_one> ::=
    select_one { <guarded_logic_clauses> }
<guarded_logic_clauses> ::=
    <guarded_logic_clause>
    | <guarded_logic_clauses> <guarded_logic_clause>
<guarded_logic_clause> ::=
    <goal> :- <guarded_goal_list> .
<guarded_goal_list> ::=
    <guard> "!" <goal_list>
<goal_list> ::=
    <goal>
    | <goal_list> , <goal>
<guard> ::=

```

<goal> <goal_list>

The intended interpretation is that a **select_one** expression has at most one clause among its **<guarded_logic_clauses>** that may be selected for reduction of the current goal. Every clause in the **select_one** has a guard, and those guards are evaluated in a coordinated fashion across the **select_one** set. If one makes the assumption of Onai, et. al. [29], that only bounded (and modest) resources are required for computation of guards, coordination is not necessary, since non-selected guards run to completion (they are not killed when another guard succeeds).

A **<logic_clauses>** expression represents a set of elements (clauses and **select_one** expressions), each of which may be selected to reduce the current goal. Members of the **<logic-clauses>** set may or may not be guarded, but all guards are interpreted independently of the other elements.

An alternate syntax would use COND-like structures in place of **select_one** expressions. The formal arguments in the head of the clause would be distinct logical variables, and the body of the clause would be a multi-way conditional, with "unpacking" of the formal arguments done explicitly. The Juniper-B1 *if-then-else()* provides an example of this form of syntax.

Neither proposal has a connection to the syntax of first-order logic. In the long run, I have preference for the former syntax -- however, I think it is useful to use the latter at present so that computations present in guards are made explicit.

Extant proposals for guards

I know of three proposals for non-deterministic logic-programming guards. Although we have already rejected non-deterministic guards, the definition of these guards may raise issues or provide solutions to some of the problems we are attempting to address.

The PARLOG language [6, 8, 9] is a parallel logic-programming language which has undergone significant modifications (to weaken it) in an attempt to make the compilation more tractable. Restriction of the (parallel) guards is a major focus of those efforts.

Concurrent PROLOG and Flat Concurrent PROLOG are described in [1, 25, 35]. [35] is the standard Concurrent Prolog definition. [25] defines a severely limited subset of CP which is thought to be compilable. [1] is supposed to provide a method for translating a subset of Concurrent Prolog to flat Concurrent Prolog. The non-deterministic guards play a major role in complicating the compiler implementation of Concurrent Prolog.

Ueda's GHC (Guarded Horn Clauses) [41, 42] is a proposal to restrict guards in order to simplify their compilation. [More to follow.]

Deterministic selection in rewriting systems

Equational programming languages [28] and term-rewriting systems must also address the issue of deterministic equation selection for reduction. I have not yet investigated the approaches they employ (although I believe O'Donnell's is syntactic).

A.5.2. Other control constructs

IC-PROLOG control constructs

Naish's MU-PROLOG control construct

Naish [26] proposes a control construct similar to, and more general than (?), PARLOG's read-only variables. For any predicate in the database, a *wait()* directive can be asserted that indicates which formal arguments must be bound before reduction of that predicate may proceed.

The *wait()* directives are similar in character to mode declarations permitted in DEC-10 PROLOG [2]. The difference is quite significant, however: whereas DEC-10 PROLOG fails when a variable argument in a goal is unexpectedly unbound, IC-PROLOG and MU-PROLOG block the reduction until some other computation binds the variable argument.

One issue this construct attempts to address is the efficiency of evaluating goals in the body of a clause, i.e., it allows an ordering of the evaluation such that "producer" terms are evaluated before "consumer" terms. In particular, it allows predicates to be restricted to functions, an important facility we are providing by a more general mechanism.

Naish's principal motivation, however, is demonstrated by the following simple problem: a database

```
append3(A,B,D,E) :-  
    append(A,B,C), append(C,D,E).  
append([],A,A).  
append(F.A,B,F.C) :-  
    append(A,B,C).
```

is defined for the *append3()* relation. In the declarative view of PROLOG programs, it should be possible to type in a goal such as

```
append3(X,[3],[4],[1,2,3,4]).
```

and have the interpreter return a binding for X such that *append3()* holds. This is

indeed what a conventional interpreter does — however, if asked for a second solution, it generates and tests a sequence of possible bindings for X of progressively longer length, with the result that a query for the second binding does not terminate.

Naish argues that usage of *cut()* prevents this situation from occurring at the expense of correctness, in particular completeness with respect to the relational interpretation of *append3()*. He claims that reordering of clauses cannot prevent the behavior, and no use of meta-logical primitives such as *var()* and *nonvar()* provides a straightforward solution in all cases.

His solution is use the following two directives (+ signifies an argument to be imported, and - signifies an argument to be exported):

```
wait(append(+,+,-)).  
wait(append(-,-,+)).
```

together with the block-until-instantiated interpretation to assert that *append* acts only to concatenate existing lists (first two arguments) into a result (third argument), or to split an existing list (third argument) into complementary sub-lists (first two arguments).

Note, however, that

- (a) These directives also affect correctness with respect to the relational interpretation of *append()*.
- (b) The blocking of reductions affects correctness because some arguments to a predicate may never become fully grounded.
- (c) The way in which *wait()* affects correctness may not be immediately evident to the programmer thinking about the declarative semantics of the program, e.g., which cases of the declarative interpretation of *append3()* are disabled by the declarations above. More generally, the semantics becomes very much more difficult in the case where *append()* is deeply nested inside of other clauses (which may also be constrained by *wait()*).
- (d) There was nothing incorrect about the behavior of the original construct, including non-termination of the search for the second binding. What the example represents is inappropriate use of existing control constructs for Naish's intended purpose. A solution that behaves as desired in the example given above, and in others Naish cites as problematical, is given by

```
append3([],B,C,D) :-  
    append(B,C,D).  
append3([F|A],B,C,[F|D]) :-
```

```

    append3(A,B,C,D).
    append([],B,B).
    append([F|A],B,[F|C]) :-
        append(A,B,C).

```

Of course, if one wishes to take seriously the role of logic-programming as a declarative language, then the non-equivalence of the two solutions is also an issue to be addressed. If one does wish to take that seriously, however, the issue should be addressed more broadly, and it is clearly not going to be resolved by adding read-only declarations.

Equivalence of read-only variables and guarded clauses

I'm 95% certain that read-only annotations and guarded clauses are equivalent in the sense that the effects of one can be achieved by the other in a straightforward fashion. It is possible that deterministic and non-deterministic cases need to be addressed separately. More to follow.

A.5.3. Addition of the *call()* predicate/operator

Implementation of *call()*

call() is heavily used in many PROLOG programs for a variety of purposes, including implementation of *bagoff()*, *setoff()*, *not()*, and other meta-logical operators. Our approach to implementing *call()* is to have it invoke the functional component, which recursively invokes the logic-programming component via the set-generator construct with *call()*'s argument as the goal. More details are provided above in the section Implementation of *call()* and *fail()*

A more powerful alternative: Kahn's *collect()*

[Kahn's *collect()* [18]]

A.5.4. Unreconstructed meta-operators

functor()

functor() is problematic because it allows application-splitting(?)

A.5.5. *assert()* and *retract()* operations

We probably want to abandon those effects of *assert()* and *retract()* that are related to database serialization, which is to say that their only effects would be to insert or delete members of a `<logic_clauses>` set as defined in OR-parallel selection of clauses.

I am in favor of abandoning *assert()* and *retract()* altogether. In a highly parallel system, a goal is reduced with respect to a database by concurrent reduction or anticipatory reduction using all clauses selected from the database. *assert()* and *retract()* affect this in two ways. First, their actions at run-time they may cause certain of these computations to be abandoned, others to be recomputed, and yet others to be initiated. Second, since a database may be shared by several concurrent reduction processes, the availability of *assert()* and *retract()* greatly complicates the semantics of synchronisation among the procedures having read access to a database, and similarly complicates the implementation. This is just another instance of the more-than-single-assignment problem in concurrent systems, which we are disposed to avoid at any reasonable cost.

A.5.6. Explicit logic databases

An alternative to the use of *assert()* and *retract()* in Juniper is the inclusion of explicit logic databases. For example, the functional-language call to the logic-programming language might be augmented to include explicit reference to a database or databases, as in the set generator expression

$$\{ \langle ?A, ?B \rangle \mid P(?A, f(?B)) \leftarrow \text{Database1} \}$$

Addition of explicit databases to the logic-programming component might require nothing other than the addition of an extra argument to the *call()* operator to name the database or databases. This sort of facility is provided by Weyhrauch's FOL [43].

This capability, along with the ability to create, merge, and intersect databases, may well serve as a reasonable substitute for the *assert()* and *retract()* operations. Moreover, if a logic-programming database was simply a type of set in a future version of Juniper, all of the database operations could be provided as library routines, rather than as additional constructs in the language.

[issues for reduction policy]

A.5.7. Negation by failure

There is no direct way of proving a negative literal NOT *p*(...) using Horn clauses and linear resolution. The usual approach to including some form of negation in Horn-clause theorem provers is to use negation-by-failure, which is a well-known and well-understood part of most PROLOG implementations [5].

The problem

In some views, PROLOG's *not()*, widely implemented as negation-by-failure, does not capture a satisfactory notion of failure [7, 26]. The naive implementation of *not()* uses *call()* in conjunction with an IF-THEN-ELSE or *cut()* construct:

```
not(?X) :-  
    call(?X), !, fail.  
not(?X).
```

or

```
not(?X) :-  
    call(?X) -> fail;  
    true.
```

The normal interpretation of a query

?- p(?X).

is, "Is there a value for X such that p is true of X (and what is it)?" which is written

$$\exists ?X \text{ } p(?X)$$

in first-order logic. What the query

?- not(p(?X))

computes, however, is, "Is there not an X such that p is true of X?" which is to say, "Succeed only if there is no X such that p is true of X." Logically this is expressed by

$$\text{NOT } \exists ?X \text{ } p(?X)$$

which is equivalent to

$$\forall ?X \text{ NOT } p(?X)$$

rather than the putatively-desired "logical" interpretation,

$$\exists ?X \text{ NOT } p(?X)$$

An alternate proposal

Clark and McCabe [7] and Naish [26] have both objected to the use of negation-by-failure on the grounds that it does not capture the logical notion of negation. Their solution (in IC-PROLOG and MU-PROLOG) to the problem of non-equivalence between the two interpretations has been to implement a restricted form of negation: the goal fails if it would fail under *both* negation-by-failure and the "logical" interpretation discussed above. That is, $\text{not}(P)$ succeeds only in those cases where

$$\forall X \text{ NOT } P$$

is equivalent to

$$\exists X \text{ NOT } P$$

which are precisely the cases in which X does not appear in P , for any variable X , and P is unprovable. That is, P must be an unprovable, grounded formula. Clark and McCabe, and Naish, are able to capture this interpretation by associating a *read-only* or *wait()* imperative with the argument of the *not()* operator, so that reduction of the goal $\text{not}(P)$ is blocked until P is fully instantiated.

This solution is unsatisfactory for two reasons:

- (i) The argument of *not()* may never become fully grounded, and hence the goal may never be reduced. This is generally inconsistent with a logical (declarative) interpretation.
- (ii) The solution is less expressive than negation-by-failure. Neither the conventional interpretation nor the proposed interpretation produces bindings as a side-effect, hence both operate as predicates. The difference in expressiveness is due to the fact that presence of a variable in the argument to *not()* is equivalent under negation-by-failure to quantification over all terms that might appear in that position.

For example, if one is trying to determine if a system is operational, one (but not the only) sufficient criterion would be that no component of the equipment is off-line. This could be encoded in PROLOG using the clause

```
operational(System) :-  
    not(status(System,Component,off-line)).
```

If the argument to *not()* were required to be grounded, the clause would instead have to generate and test a complete list of components, a procedure which might be lengthy or impossible (e.g., "This neighborhood is safe if there have been no

crimes reported here in the past year" -- but the universe of potential crime victims in such reports is unknowably large).

A solution for Juniper

Our solution is to include negation-by-failure in Juniper. It is attainable, useful, and well-understood. The answer to objections put forth by IC-PROLOG and MU-PROLOG proponents may be to change the name of *not()* to *fail()*.

A.6. Definition of Juniper-C

Juniper-C is conceived of as including full functions.

A.6.1. Addition of full functions in the logic-programming component

Control over full function calls from the logical language will be more difficult than for pure functions calls, since it incorporates full unification, and, potentially, higher-order unification.

The loss of referential transparency associated with logical variables appearing in full functions is also a major concern, although there may be interpretations of full functions under which referential transparency is not given up.

There exist (at least) five proposals for adding full functions within the logic-programming paradigm. I will provide more detailed and useful summaries in a future iteration of this paper.

Kahn [16, 17]: The general strategy is very elegant: the unifier is augmented by adding a facility for equality reasoning. Function definitions appear as equality assertions (with a left-to-right interpretation), and their reductions are treated as part of the equality-reasoning mechanism. The unification of logical variables as part of the functional reduction mechanism is handled very naturally by a recursive call to the unifier. The control strategy for selection of unification-directed equalities for reduction is breadth-first (shortest path), although one is referred to the second paper for discussion of the details of this approach. A very nice approach, in any case, but no conclusive results are presented.

Reddy [32, 33] introduces *narrowing*, first introduced in work on theorem-proving for equational theories [12, 14, 22, 37], as an operational semantics for these languages. This is the most comprehensive approach to full functions that I have seen, and presents a rather complete discussion of its semantics, including extensions to lazy narrowing and possible applications in higher-order unification and for parallel implementation.

Subrahmanyam and You [38, 39]: The papers are virtually identical. The work is not unlike Reddy's in spirit, but suffers from several difficulties. Among these is the failure to motivate several decisions (goals can be reduced in order to obtain a unifiable term, but terms in the heads of function definitions may not) that results in incompleteness, and the assumption that functions can be tested for equality. Nevertheless, it should be worthwhile to identify the issues it chooses to address, and compare the solutions with those arising from the other two proposals.

Darlington, Pull, and Field [10] propose a unified functional and logic language based on the reduction model. [Further discussion]

Goguen and Meseguer [14]

A.7. Semantics of Juniper

The semantics of pure PROLOG has been studied at some length [23], as have the semantics of functional programming [15]. An important part of logic-programming languages that is not a part of pure PROLOG, and which is especially important to Juniper, is the *call()* predicate. Perlis [31] has provided a model based on work by Kripke and Gilmore [13, 20] which I believe is a sufficient model for pure PROLOG with *call()* and negation-by-failure added. This would provide a good starting point for providing a semantics for Juniper.

References

1. Bloch, C., Source-to-source transformations of logic programs, Tech. Rep. CS84-22, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1984.
2. Bowen, D. L., Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D., Decsystem-10 Prolog User's Manual, Occasional Paper 27, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, November, 1982.
3. Burstall, R. M., Writing Search Algorithms in Functional Form, *Machine Intelligence 3*, (1968), 373-385, Edinburgh Univ. Press.
4. Carlsson, M., On Implementing Prolog in Functional Programming, *New Generation Computing 2*, (1984), 347-359, Ohmsha, Ltd. and Springer-Verlag.
5. Clark, K. L., Negation as Failure, in *Logic and Data Bases*, H. Gallaire and J. Minker (ed.), Plenum Press, New York, New York, 1978, 293-324.
6. Clark, K. L. and Gregory, S., A Relational Language for Parallel Programming, *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, N.H., October 1981, 171-178.
7. Clark, K. L., McCabe, F. G. and Gregory, S., IC-Prolog language features, in *Logic Programming*, K. L. Clark and S. Tarnlund (ed.), Academic Press, London, 1982, 253-266.
8. Clark, K. L. and Gregory, S., PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, Imperial College of Science and Technology, London, 1983.
9. Clark, K. L. and Gregory, S., PARLOG: A Parallel Logic Programming Language, Research report DOC 84/4 (revised), Imperial College of Science and Technology, London, June, 1985.
10. Darlington, J., Field, A. J. and Pull, H., The unification of functional and logic languages, Research report DOC 85/3, Department of Computing, Imperial College of Science and Technology, 1985.
11. Dwork, C., Kanellakis, P. and Mitchell, J., On the Sequential Nature of Unification, *Journal of Logic Programming 1*, 1 (1984), 35-50.
12. Fay, M., First-order Unification in an Equational Theory, in *Proceedings, Fourth Workshop on Automated Deduction*, Austin, Texas, February, 1979, 161-167.
13. Gilmore, P., The consistency of partial set theory without extensionality, in *Axiomatic Set Theory*, T. Jech (ed.), American Mathematical Society, Providence, R. I., 1974, 147-153.
14. Goguen, J. and Meseguer, J., Equality, Types, Modules and (Why Not?) Generics for Logic Programming, *The Journal of Logic Programming 1*, 2 (1984), 179-210.
15. Gordon, M., *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

16. Kahn, K. M., Uniform: A Language Based Upon Unification Which Unifies Much of LISP, Prolog, and Act 1, in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981.
17. Kahn, K. M., The implementation of Uniform: a knowledge-representation/programming language based upon equivalence of descriptions, Technical report 9, Uppsala Programming Methodology and Artificial Intelligence Laboratory, Department of Computer Science, University of Uppsala, 1982.
18. Kahn, K. M., A Primitive for the Control of Logic Programs, in *Procs. 1984 International Symposium on Logic Programming*, IEEE, February, 1984, 242-251.
19. Kornfeld, W. A., Equality for Prolog, in *Proceedings, Seventh International Joint Conference on Artificial Intelligence*, A. Bundy (ed.), William Kaufmann Inc., 1983, 514-519.
20. Kripke, S., Outline of a Theory of Truth, *The Journal of Philosophy* 72, (1975), 690-716.
21. Landin, P. J., The Next 700 Programming Languages, *Communications of the Association for Computing Machinery* 9, (1966), 157-164.
22. Lankford, D. S., Canonical Inference, ATP-32, University of Texas at Austin, 1975.
23. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1985.
24. Martelli, A. and Montanari, U., An efficient unification algorithm, *ACM Trans. Prog. Lang. and Systems* 4, 2 (April, 1982), 258-282, ACM.
25. Mierowsky, C., Design and Implementation of Flat Concurrent Prolog, Tech. Rep. CS84-21, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1984.
26. Naish, L., An Introduction to MU-Prolog, Technical Report 82/2, Department of Computer Science, University of Melbourne, March 1982 (Revised July 1983).
27. Nilsson, M., The world's shortest Prolog interpreter?, in *Implementations of PROLOG*, J. A. Campbell (ed.), Ellis Horwood Ltd., Chichester, 1984, 87-92.
28. O'Donnell, M. J., *Equational Logic as a Programming Language*, MIT Press, Cambridge, 1985.
29. Onai, R., Aso, M., Shimizu, H., Masuda, K. and Matsumoto, A., Architecture of a Reduction-based Parallel Inference Machine: PIM-R, *New Generation Computing* 3, 2 (1985), 197-228, Ohmsha, Ltd. and Springer-Verlag.
30. Paterson, M. S. and Wegman, M. N., Linear Unification, in *Proceedings of the 8th ACM Symposium on the Theory of Computing*, 1978, 158-167.
31. Perlis, D., Languages with Self-reference I: Foundations, *Artificial Intelligence* 25, 3 (March, 1985), 301-322.
32. Reddy, U. S., Narrowing as the Operational Semantics of Functional Languages, in *Procs. 1985 Symposium on Logic Programming*, IEEE, 1985, 138-151.

33. Reddy, U. S., On the relationship between functional and logic programming, in *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom (ed.), Prentice-Hall, 1986, 3-36.
34. Robinson, J. A., *New Generation Knowledge Processing/Syracuse University Parallel Expression Reduction: First annual progress report*, Syracuse University, December, 1984.
35. Shapiro, E. Y., A subset of Concurrent Prolog and its interpreter, Tech. Rep. 003 (revised February 1983), ICOT - Institute for New Generation Computer Technology, Tokyo, Japan, January, 1983.
36. Siekmann, J. H., Universal Unification, in *Procs. Seventh International Conference on Automated Deduction*, Springer-Verlag, 1984.
37. Slagle, J. R., Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity, *Journal of the ACM* 21, 4 (October, 1974), 622-642.
38. Subrahmanyam, P. A. and You, J., Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming, in *Procs. 1984 International Symposium on Logic Programming*, IEEE, Atlantic City, N.J., 1984, 144-153.
39. Subrahmanyam, P. A. and You, J., Pattern Driven Lazy Reduction: a Unifying Evaluation Mechanism for Functional and Logic Programs, in *Proceedings, 11th Symposium on Principles of Programming Languages*, ACM, 1984, 228-234.
40. Turner, D. A., *SASL Language Manual*, University of St. Andrews, 1976.
41. Ueda, K., Guarded Horn Clauses, Tech. Rep. 103, ICOT - Institute for New Generation Computer Technology, Tokyo, 1985.
42. Ueda, K. and Chikayama, T., Concurrent Prolog Compiler on Top of Prolog, in *Procs. 1985 Symposium on Logic Programming*, IEEE, 1985, 119-126.
43. Weyhrauch, R. W., Prolegomena to a Theory of Mechanized Formal Reasoning, *Artificial Intelligence* 13, 1 (1980), 133-170, North-Holland.

Appendix A-A: Syntax of Juniper-B1

```

<program> ::= <expr> ? | DEF <defs> ? | DEC <logic_clauses> ?
<expr> ::= <expr> WHERE <defs> | <condexp>
<condexp> ::= <opexp> -> <condexp> ; <condexp> | <listexp>
<listexp> ::= <opexp> , ... , <opexp> | <opexp> , | <opexp>
<opexp> ::= <prefix> <opexp> | <opexp> <infix> <opexp> | <comb>
<comb> ::= <comb> <simple> | <simple>
<simple> ::= <name> | <constant> | ( <expr> ) | <sfexpr> | <logic_expr>

<defs> ::= <clause> ; <defs> | <clause>
<clause> ::= <namelist> = <expr> | <name> <rhs>
<rhs> ::= <formal> <rhs> | <formal> = <expr>
<namelist> ::= <struct> , ... , <struct> | <struct> , | <struct>
<struct> ::= <formal> : <struct> | <formal>
<formal> ::= <name> | <constant> | ( <namelist> ) | ( <appl> )
<appl> ::= <name> <formal> | <appl> <formal>

<constant> ::= <numeral> | <charconst> | <boolconst> | () | <string>
<numeral> ::= <real> <scale-factor> | - <real> <scale-factor>
<real> ::= <digit>* | <digit>* . <digit>
<scale-factor> ::= e <digit>* | e - <digit>*
<boolconst> ::= TRUE | FALSE
<charconst> ::= % <any char> | SP | NL | NP | TAB
<string> ::= '<any message not containing unmatched quotes>'

<sfexpr> ::= { <expr> ; <qualifiers> } | { <generator> ; <qualifiers> }
<qualifiers> ::= <qualifier> | <qualifiers> ; <qualifier>
<qualifier> ::= <generator> | <filter>
<generator> ::= <name> <- <expr> | <name> <- <generator>
<filter> ::= <expr>

<logic_expr> ::= { < <logic_var_list> > | <goal_list> }
<goal_list> ::= <goal> | <goal_list> , <goal>
<goal> ::= <name> | <name> ( <term_list> )
<term_list> ::= <term> | <term_list> , <term>
<term> ::= <simple_term> | <list_term> | <name> ( <term_list> )
<simple_term> ::= <name> | <numeral> | <charconst> | <boolconst>
| <logic_var>
<list_term> ::= [] | [ <term_list> ] | [ <term> | <term> ]
<logic_var_list> ::= <logic_var> | <logic_var_list> , <logic_var>

<logic_clauses> ::= <logic_clause> | <logic_clauses> <logic_clause>
<logic_clause> ::= <goal> :- <goal_list> . | <goal> :- . | <goal> .

```

Operators (in order of decreasing binding power)

:, ++, -	infix (right associative)
..	infix (non-associative)
...	postfix
	infix
&	infix
-	prefix
>>, >, >=, =, ~=, <=, <, <<	infix
+, -	infix
+, -	prefix
*, /, div, rem	infix
**	infix (right associative)
#	prefix

Pre-defined names

abs all and append application arctan arg
 code cons converse char cjustify concat cost count
 decode digit digitval drop
 e entier eq exp
 filter foldl foldr for from function functor
 hd
 I interleave intersection iterate
 K
 lay layn length letter list listdiff ljust log logic_var logical
 map member mkset
 not number
 or
 pi plus printwidth product
 reverse rjustify
 show sin some spaces sqrt sum
 take times tl
 union until
 while
 zip

Appendix A-B: Reconstruction of meta-operators

It is important for us to know where Juniper stands relative to PROLOG in regard to meta-operators. The following lists the principal meta-operators found in CPROLOG 1.5, their intended function, and the primitive meta-operations that are sufficient for their reconstruction. This permits us to anticipate the meta-operations that will be required in Juniper to provide equivalent functionality: it appears that replacement of *call()* and *cut()*, together with the addition of explicit databases to replace *assert()* and *retract()* functionality, will suffice.

Control meta-operators

<i>Operator</i>	<i>Operation</i>	<i>Dependent upon</i>
<i>call(P)</i>	Recursive call to prove P	assumed primitive

Logical interpretation is "There exist bindings for free variables of P such that P holds." The free variables of P are suitably bound following success of *call()*.

<i>cut()</i>	PROLOG <i>cut()</i> operation	assumed primitive

Treat the current goal as complete if backtracking search occurs.

<i>not(P)</i>	Negation (sort of)	<i>call()</i> , <i>cut()</i>

Logical interpretation is "There do not exist bindings for free variables of P such that P holds." Or, operationally, "*call(P)* fails." Implemented as

not(P) :- call(P), !, fail.
not(P).

<i>P -> Q ; R</i>	if P then Q else R	<i>call()</i> , <i>cut()</i>
<i>P -> Q</i>	if P then Q	

Implemented as

if-then-else(P,Q,R) :- call(P), !, call(Q).
if-then-else(P,Q,R) :- call(R).

if-then(P,Q) :- call(P), !, call(Q).

P ; Q

P succeeds or Q succeeds

call()

Can be implemented as

or(P,Q) :- call(P).

or(P,Q) :- call(Q).

Structure meta-operators

Operator

Operation

Dependent upon

*atom(X),
number(X),
integer(X),
primitive(X)*

data-type predicates

assumed primitive

*var(X),
non_var(X),
bound(X),
unbound(X)*

variable predicates

call(), cut()

Can be implemented by

var(X) :- not(not(X=1)), not(not(X=2)).

non_var(X) :- var(X), !, fail.

unbound(X) :- var(X).

bound(X) :- unbound(X), !, fail.

*head(L,H),
tail(L,T)*

list selector functions

unification as part of
procedure call

Can be implemented by

head([H,T],H).

tail([H,T],T).

*Term =.. List,
arg(Index,Term,Name),
functor(Term,Fun,Name)*

term selector functions

unavailable in Juniper-A

Within the current scheme, logic-programming terms (and probably atomic formulae) are function-applications. Application predicates and selector functions have been added to the Juniper-A functional interpreter, and will therefore be available to the logic-programming component in Juniper-B. If a logic-programming database is treated as a type of set in Juniper-B, as the single global database is in Juniper-A, it will not be difficult to implement the full relational form of *functor()* with respect to an explicit database.

$X == Y,$	full term equivalence	<i>call()</i> , <i>cut()</i>
$X \backslash == Y$		unavailable in Juniper-A

This predicate tests for full equivalence of terms including equivalence of logical variables. It can be implemented through term decomposition and checking of variable equivalence:

```
var_equiv(X,Y) :- var(X), var(Y), not(var_equiv1(X,Y)).
var_equiv1(1,2).
```

<i>name(Name,List)</i>	convert name to char list	unavailable in Juniper-A

Takes functor name as *Name* and returns a list of the constituent characters in *List*. The function is or should be available in the functional component, and will become available to the logic-programming component in Juniper-B.

Set-construct macro-operators

Operator	Operation	Dependent upon
<i>setof()</i> , <i>bagof()</i>	set constructors	various (not available in Juniper-A)

setof() and *bagof* provide the logic-programming component with the ability to construct sets of solutions using recursive calls to itself. The set-generator construct which has been proposed as the functional-component interface to the logic-programming component has much the same character as these constructors. Since the functional component cannot be called from the logical component in Juniper-A, however, the set-constructors will not be accessible in Juniper-A.

setof() and *bagof()* depend on *var()* and *;/()*, which can be constructed from the *call()* and *cut()* primitives. In addition they depend on *==()*, *functor()*, and *arg()*, which can be defined in the functional component of Juniper-A but will not be available to the logic-programming component until function calls are provided

in Juniper-B. Finally, they depend on the ability to modify the database in the logic-programming component, a capability we do not intend to provide directly. Implementation of *bagof* and *setof()* should be a useful test of the explicit logical databases proposed in the paper as a substitute for *assert()* and *retract()*.

Appendix A-C: Proof-stream interpreter for Juniper-A

DEF

```
|||
|||
|||
|||
|||
|||
```

```
prove2()
prove2 stream goals i
```

```
prove2 () goals i
= ()
```

```
prove2 stream () i
= stream
```

```
prove2 stream (goal:rest) i
= prove2 (resolve2 stream goal i (assertions goal))
      rest
      i
```

```
|||
|||
|||
|||
|||
|||
```

```
resolve2()
resolve2(stream goal i assertions)
```

```
resolve2 () goal i assertions
= ()
```

```
resolve2 ((head:body):rest) goal i assertions
= append (resolve_1 goal i assertions body head)
  (resolve2 rest goal i assertions)
```

```
|||
|||
|||
|||
|||
|||
```

```
resolve_1()
resolve_1 goal i assertions j env
```

```
resolve_1 goal i () j env
= ()
```

```
resolve_1 goal i assertions j ()
= ()
```



```
resolve_1 goal i ((head:body):rest) j env
  = append s t
  WHERE
    s = prove2 (env_1:(j+1)) body j
    t = resolve_1 goal i rest j env
    env_1 = unify goal i head i env
```

?

Appendix A-D: Success-continuation interpreter for Juniper-A

Define interpreter for logic-programming component of Juniper-A.

DEF

provel(goals, i, continuation, environment, j)

provel() is a continuation-based prover. Choice points in the search tree are "stacked" on the continuation argument for correct continuation following AND- or OR-node processing.

goals: list of (conjunctive) goals to be solved
i: current level of variable indexing. Each variable instance is distinguished by the "level" at which it first appears. i maintains the current level.

continuation: a list (stack) of saved choice points. Alternative branches in the search tree are pushed at each choice point. The stack is popped for continuation upon completion of processing for each branch of an AND node, or for backtracking upon completion of processing for each branch of an OR node.

environment: contains current variable bindings. A null environment signals that unification of the clause head has failed.

j: next level of variable indexing. This is not always i+1 because on return to level i during processing of an AND node, levels greater than i must be maintained, so that j contains the value one greater than the maximum found in the preceding AND branches.

If null environment then unification failed; return nil.

provel goals i cont () j
= ()

///
/// If AND-goals are proven and continuation stack is empty, we
/// have success, and return current environment.
///

```
provel () i () env j  
    = (env,)
```

///
/// Else if AND-goals are all proven, pop last continuation.
///

```
provel () i cont env j  
    = cont env j
```

///
/// Resolve first goal using the clauses asserted for its head;
/// prepend remaining goals to continuation.
/// This definition implements AND-node processing in the search
/// tree. The environment and next level index will be provided at
/// the time that the continuation (next conjunct) is invoked.
///

```
provel (first:rest) i cont env j  
    = resolve1 first i (assertions first) j (provel rest i cont) env
```

///
/// resolve1(goal, i, assertions, j, continuation, environment)
///

/// goal: single goal to be resolved against assertions.
/// i: current level index.
/// assertions: list of assertions remaining for use in
/// resolution.
/// j: next level index.
/// continuation: continuation.
/// environment: binding environment in which to perform
/// reduction.
///

///
/// If no clauses to resolve against, then fail
///

```

resolve1 goal i () j cont env
  = ()

```

Implements OR-node processing. Clauses in database are indexed via tail-recursion using the same environment, continuation, and level indices. WHERE clause implements complete resolution of first disjunct.

```

resolve1 goal i assertions j cont env
  = append s t
    WHERE
      s = prove1 (tl (hd assertions)) j cont
          (unify goal i (hd (hd assertions)) j env) (j+1)
      t = resolve1 goal i (tl assertions) j cont env

```

```

unify(term1, index1, term2, index2, environment)

```

The functional language component has been augmented to include meta-predicates for recognition of unreduced applications (`application()`) and logic variables (`logic_var()`). In addition, functions have been added which return the function and argument of an application (`functor()` and `arg()` respectively), and determine structural (based on list and application constructors) equality.

An attempt to extend the MATCH operation to handle unreduced applications proved to be too problematic at this time.

`unify()` signals failure by returning the empty list.

```

unify t1 i t2 j env
  = unify_1 (ult (t1:i) env) (ult (t2:j) env) env

```

```

unify_1(t1, t2)

```

`unify_1()` performs unification of variable instances and canonical terms to create new environments from existing environments.

Once again, a null environment is reflects failure to unify.

```
unify_1 t1 t2 ()  
  = ()
```

Structural decomposition of lists for recursive unification.

```
unify_1 ((f1:rf1):r1) ((f2:rf2):r2) env  
  = unify rf1 r1 rf2 r2 (unify f1 r1 f2 r2 env)
```

Decomposition of applications for recursive unification.

```
unify_1 ((f1 a1):r1) ((f2 a2):r2) env  
  = unify f1 r1 f2 r2 (unify a1 r1 a2 r2 env)
```

Case of variables (produces new bindings)
and (grounded) equality

```
unify_1 (f1:r1) (f2:r2) env  
  = logic_var f1 -> ((f1:r1):(f2:r2)):env  
    logic_var f2 -> ((f2:r2):(f1:r1)):env  
    (f1=f2) -> env  
    ()
```

ult(name,environment)

ult() chains through name/value pairs in the environment
to get a canonical value for its first argument.

```
ult name env  
  = null pair -> name  
    (name = value) -> name  
    ult value env
```

```
pair = assoc name env
(term: value) = pair
```

assoc() returns the first name/value pair in list whose name matches the first argument to **assoc()**.

Appendix B

Small Logic Programs to Test Juniper

by

**Donald P. McKay
Keith Cassell
Judy Pohl Clark**

Table of Contents

Section	Title	Page
B.1	Introduction	B-1
B.2	Programs	B-3
B.2.1	reverse	B-4
B.2.2	gram2	B-4
B.2.3	peano	B-5
B.2.4	facto.peano	B-5
B.2.5	fibonacci	B-5
B.2.6	qsort	B-6
B.2.7	geo	B-6
B.2.8	foxes	B-7
B.2.9	sets	B-8
B.2.10	fibonacci	B-8
B.2.11	queens	B-8
B.2.12	diff	B-8
B.2.13	bubble sort	B-9
B.2.14	findall	B-9
B.2.15	maplist	B-9
B.3	Juniper Translation Difficulties	B-9
B.3.1	setof	B-9
B.3.2	maplist	B-10
B.3.3	Overloading of function names	B-10
B.3.4	The symbolic differentiation program	B-11
B.4	Acknowledgements	B-12
B.5	Summary	B-12
B.6	References	B-13
B.7	Code listings	B-14
B.7.1	Reversel	B-14
B.7.2	gram2.pl	B-15
B.7.3	peano.pl	B-18
B.7.4	facto.peano.pl	B-20
B.7.5	fibonacci.pl	B-21
B.7.6	qsort.pl	B-22
B.7.7	geo.pl	B-23
B.7.8	foxes.pl	B-25
B.7.9	sets.pl	B-27
B.7.10	fibonacci.pl	B-29
B.7.11	queens.pl	B-30

B.7.12	queens.pl	B-32
B.7.13	diff.pl	B-34
B.7.14	diff.pl	B-37
B.7.15	bubble.pl	B-41
B.7.16	bubble.pl	B-42
B.7.17	maplist.pl	B-43
B.7.18	maplist.pl	B-44
B.7.19	findall.pl	B-45
B.7.20	findall.pl	B-46

Appendix B

Small Logic Programs to Test Juniper†

B.1. Introduction

The MASC project is developing a programming language named Juniper that combines features of both functional and logical languages. The Juniper language will eventually run on a parallel machine, although it is presently being developed on conventional machines, e.g., VAX and SUN.

The Juniper language will change as the project reaches maturity. The present version of the language, Juniper-B1, is under active development[1,2]. The logical component of Juniper is a simplified Prolog dialect. The features of the logical component of Juniper-B1 relative to Prolog need to be examined to determine 1) whether current Logic-Based Systems applications or translations of them are definable within Juniper-B1, and, 2) whether the logic component is complete enough to capture a rich set of Prolog programs. This report describes a part of the effort to develop applications for Juniper which focus on the logic component and its interaction with the functional component. A part of the applications effort focused on the functional component is described elsewhere.

The purpose of these programs is to test various features of logic programming. In particular, the programs are designed to test those features of logic programming that set it apart from functional programming, as well as the interface between the logic component and functional component.

Logic programming has three main characteristics. These are

- (1) Relational Notation,
- (2) Logical Variables and Unification, and
- (3) Search-based Computation.

All Prolog programs are written in relational notation as opposed to functional notation. This notation demonstrates Prolog's declarative nature. It allows the expression of relationships among arguments in a general manner such that a logic procedure, i.e., the set of clauses defining a relation, is *directionally transparent*. For example, the append relation can be defined by the following logic procedure:

```
append([], X, X).  
append([X1 | Rest], Y, [X1 | Z]) :- append(Rest, Y, Z).
```

This definition of append describes the relationship among two lists and their concatenation. It can be used to compute what traditionally have been considered many different functions. Some of these uses include:

† This section is also available as LBS Technical Memo 33.

- (1) A "predicate" which tests if two lists can be concatenated to form a third as in
`append([1,2,3], [4,5,6], [1,2,3,4,5,6]).`
- (2) A "function" which concatenates two lists as in
`append([1,2,3], [4,5,6], X).`
- (3) A "function" which generates a collection of lists which when concatenated are a given list as in:
`append(X, Y, [1,2,3,4,5,6]).`

It also makes the representation of a relational database in Prolog very natural. An example of this can be seen in a later program which implements a small geographic database.

Logical variables and unification are also used in all Prolog programs. This feature of logic programming serves the same purpose as parameter passing and value return in functional languages. This can be seen in all of the example programs. Logical variables also make it possible to build very general data structures by allowing pieces of the structures to be uninstantiated. This is a characteristic of tree building in Micro-Pundit which will be presented in a separate paper[3].

Search-based computation allows logic programs to follow a path through a search space with a certain instantiation of the logical variables found along that path, and if it does not lead to a solution, it has the ability to take another path. In conventional Prolog systems this is implemented via backtracking. This is exemplified in two programs (foxes and queens). Both have more than one path to a solution, but will do quite a bit of search before finding one.

The overall approach to developing programs for Juniper is to produce two basic sets of programs: 1) benchmark programs for test, validation and performance analysis and larger applications of interest. The specific approach taken is a phased implementation effort in which the elaboration of a benchmark set for the logic component will result in a representative sample of programs first. However, the set of programs will be extended to include more programs for thorough benchmarking and testing of Juniper implementations. Finally, a validation set of programs is being developed separately from the benchmark and test set. It is intended that all of the programs described in this paper test the logic component. Issues such as analysis and measurement of available parallelism or how to compose programs in *idiomatic Juniper* are being addressed separately.

The benchmark programs are intended to be small programs to demonstrate both the Juniper language and implementations of Juniper. The desired features of the benchmark programs include:

- (1) the ability of the benchmark to test and measure the performance of Juniper;
- (2) the ability of the benchmark to elucidate the language coverage of Juniper for simple Prolog programs;

- (3) the acceptance of the benchmark as a representative program, i.e., its inclusion by others in benchmark sets.

In order to test and measure the performance of Juniper it is important that a candidate benchmark be scalable to run larger or smaller sample sizes as appropriate to the test. Another preferred feature of a benchmark candidate is that it rely on a small set of language features in order to minimise interference effects for benchmarking and to be able to test individual language features. Finally, there must be a reasonable expectation that a candidate program be translatable to Juniper, i.e., uses the primitives defined in Juniper-B1 or language predicates/functions which could be reasonably defined. Thus, programs which made extensive use of Prolog's assert/retract were not considered as candidates for this first set since there were no obvious direct translation methods other than global addition of arguments to predicates. Also, it is imperative that comparative studies be able to be performed.

The larger applications are being developed in a staged approach in which initial versions of the programs are developed in pure Prolog and subsequently rewritten using features unique to Juniper, e.g., MicroKNET[4] and MicroPundit[3]. In either case, a concern is that the programs have reasonable chance of being runnable programs, e.g., produce reasonable outputs for reasonable inputs and terminate. In order to assure this, these programs were written in pure Prolog first.

This paper describes an initial set of programs for testing and benchmarking purposes. The range of programs includes logic programs which implement naive reverse, factorial, peano arithmetic and "database" query. This set also includes some programs which are part of the standard set of Prolog benchmark programs which SDC and others use to test and benchmark Prolog systems[5]. Some of the programs were obtained from a standard Prolog library. Other of the programs are variants produced to test specific features of the Juniper language both from a syntactic and from an operational point of view.

Some of the programs present translation difficulties; the resolution of these is still under discussion. The major sources of concern are 1) including logical variables in calls from the logic component to the functional component, 2) mapping the Prolog setof construct into Juniper-B1's set expression bridging constructs, and 3) determining which function symbols will cause evaluation and which will not. Another part of the applications development for Juniper is investigating larger applications such as a knowledge representation system and a natural language system and how to exploit the relational and functional components of Juniper.

B.2. Programs

The programs presented in this paper exist as Prolog programs and as Juniper programs. The reasons they exist in both forms is to provide a medium in which to develop and debug programs (Prolog) and to serve as the basis for comparison of their Prolog and Juniper instantiations. To this end, where the source for the Prolog and Juniper versions differ significantly, both versions are contained in Appendices. The Prolog version and Juniper version differ significantly when more than defining forms

and comment characters differ.

Most of programs discussed in this paper are simple programs that require, in the Prolog version, no *extralogical* features unless otherwise noted. In particular, these programs do not use:

- 1) predicates that add or remove items from the database, e.g., `assert/retract`;
- 2) predicates that artificially prune the search tree, e.g., `cut`;
- 3) predicates that construct or dissect terms, e.g., `=..`, `name`;

The elements of these programs deserving comment are:

Many programs use the Prolog predicate `\+` which implements negation as failure. These usages are difficult to avoid, particularly when the use of `cut` is disallowed.

Some programs use the built-in predicate `is` in order to assign a value. In all cases encountered, the occurrence of `is` can be replaced by `=`. However, if the functional expression is copied/moved within the source or is not executed in the given linear order, ensuring that all of the logic variables are bound at the time of functional reduction may dictate that functional expressions appear in the body of clauses rather than in the head.

One program uses `setof` in its sample queries. Some uses are problematic in Juniper and must be translated to other forms.

One program uses function symbols which may be confused with arithmetic operators, i.e., the function symbol `+` could appear as the functor of a term which was not intended to be functionally reduced.

B.2.1. reverse

This program reverses the top-level elements of a list using `append`. The definition used here is typically referred to as *naive reverse* and is a standard Prolog benchmark. To reverse a list, type:

```
?- rev(List, Reversed_List).
```

where `List` is the list to be reversed.

B.2.2. gram2

This grammar parses a small set of English sentences. Some sentences may generate more than one parse. To parse a sentence, type:

?- sentence(Parse_tree, Sentence, Remainder)

where Parse_tree is the parse tree produced by Sentence, and Remainder is that part of the end of the sentence that is not involved in the parse.

sample calls:

?- sentence(X, [john,was,believed,to,have,been,shot,by,mr_nose], []).
?- sentence(X, [was,john,shot], []).
?- sentence(X, [fred,likes,mary], []).

B.2.3. peano

This program does integer arithmetic and arithmetic comparisons without using **is**. Numbers are represented as follows:

A zero is represented as **0**.

Positive numbers correspond to the number of "s"s to the left of **0**, e.g. **s(0)** represents 1; **s(s(0))** represents 2; **s(s(s(0)))** represents 3, etc.

The following operators are implemented: **add**, **minus**, **mult**, **div**, **mod**, **power_of**, **greater**, **less**, and **equals**.

B.2.4. facto.peano

This program uses Peano arithmetic to calculate factorials (e.g., the factorial of 0 is 0, the factorial of 1 is **s(0)**, etc.). To calculate the factorial of a number, type:

fact(N, Fact_N)

where Fact_N is the factorial of N, e.g. **fact(3,6)**. Because this program uses Peano arithmetic, **fact(3,6)** is actually **fact(s(s(s(0))), s(s(s(s(s(0)))))))**

Given N, the program generates Fact_N and quickly fails on a second attempt. Given Fact_N, the program generates N but looks for a second answer for a long time if requested to do so.

B.2.5. fibo.peano

This program uses Peano arithmetic (e.g. 0 is 0, 1 is **s(0)**, 2 is **s(s(0))**, 3 is **s(s(s(0)))**, etc.) to calculate members of the Fibonacci series. To use the program, type:

fibo(Nth,N)

where N is the Nth Fibonacci number, and the 0th and 1st Fibonacci number is 1.

Given Nth, the program generates N and quickly fails on a second attempt. Given N, the program generates Nth but looks for a second answer for a long time if requested to do so.

B.2.6. qsort

This quick sort sorts numbers in ascending order and does not remove duplicates. To sort a list of numbers, type:

```
?- qsort(Orig, Sorted, []).
```

where Orig is the original, unsorted list and Sorted is that list sorted. For convenience, one can type:

```
?- sort_list(Orig, Sorted).
```

which sorts a previously prepared 50 element list.

B.2.7. geo

This program consists mostly of a database of geographic facts represented as a collection of logic ground clauses. Because most of the database is composed of predicates that have more than 150 clauses, there is ample opportunity to exploit *database or-parallelism*. A number of sample query predicates in Prolog are provided below. Notice that some of the sample queries require *setof*.

What is a city in Japan?

```
db1(S):- country(S, japan).
```

What are the cities in Japan?

```
db2(S) :- setof(C, country(C,japan), S).
```

Which far-east country is landlocked?

```
db3(C) :- region(C,far_east), iscountry(C), landlocked(C).
```

List all the far-east countries which are landlocked.

```
db4(S) :- setof(C, db3(C), S).
```

Which country borders two seas?

```
db5(C) :- iscountry(C), setof(X, border_sea(C,X), S), size(S,2).
```

Which countries border two seas?

```
db6(T) :- setof(C, db5(C),T).
```

Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of the USA?

```
db7(C) :-  
    population(usa,Y), borders(C,mediterranean_sea), iscountry(C),  
    borders(C,C1), iscountry(C1), borders(C1,C2),  
    population(C2,X), X>Y.
```

Which countries bordering the Mediterranean border a country that is bordered by a country whose population exceeds the population of the USA?

db8(S) :- setof(C, db7(C), S).

What is the oil production of countries which are members of OPEC but not the Arab League?

db9(S) :- setof([C,P], oil_production(C,P), S).

oil_production(C,P) :-

belongs(C,opec), \+belongs(C,arab_league), produces(C,oil,P,1975).

Which is the ocean that borders African countries and that borders European countries?

db10(X) :-

open_water(X), borders(X,C), region(C,africa), iscountry(C),

borders(X,C1), region(C1,europe), iscountry(C1).

Which countries belong to the organization Org?

db11(Org,S) :- setof(C, belongs(C, Org), S).

What are the organizations to which a country may belong and which countries belong to each?

db12(S) :- setof([Org,Countries], setof(C, belongs(C,Org), Countries), S).

Some variants of the above setof are not translatable directly to Juniper-B1 using the set expression bridging construct. The difficulty results from the importation of (unbound) logic variables into the functional component. For example, in db5 if iscountry is not included in the clause then the logic variable C appears free in the subsequent setof expression. A similar problem exists in db12.

B.2.8. foxes

This program solves the following puzzle:

The Farmer has a goose, grain, and a fox. He needs to cross a river, but his boat will only hold himself and one other animal. The problem arises because 1) the grain and the goose cannot be left unattended, nor 2) can the goose and the fox because of their dietary habits.

The problem solver keeps track of the current state in a term of the form:

state(Farmer,Fox,Goose,Grain).

Each creature can take on a value of either 1 or 2, each number representing a side of the river.

To generate a solution, type:

```
?- path( state(1,1,1,1), state(2,2,2,2), Solution).
```

There are two solutions to this query.

B.2.9. sets

This program provides a number of operations on sets. A set is represented as an unordered list of unique elements. An example of each operation follows.

```
?- member(X,Y). Is X a member of the set Y?
```

```
?- subset(X,Y). Is X a subset of Y?
```

```
?- union(X,Y,Z). The union of X and Y is Z.
```

```
?- intersection(X,Y,Z). The intersection of X and Y is Z.
```

```
?- subtract(X,Y,Z). Z is the result of subtracting Y from X.
```

```
?- seteq(X,Y). Do the sets X and Y contain the same elements?
```

B.2.10. fibo

This program also computes the members of the Fibonacci series, however, it uses conventional arithmetic notation instead of Peano arithmetic. This program uses `is`, so it may be considered as tainted by functional constructs. As in `fibonacci.pl`, to get the Nth member of the series, type:

```
?- fibo(Nth,N).
```

B.2.11. queens

This program provides solutions for the "n-queens problem" which states that N queens must be placed on an NxN chessboard such that no queen may attack another. To generate a solution, type:

```
?- queens(Board_size, Soln).
```

This program also uses `is`.

B.2.12. diff

This program performs some simple symbolic differentiations and will do some simplification of the result. To use this program, type:

?- ddx(X, DDX).

where X is the term that is to be differentiated. This program can differentiate terms with +, -, *, /, ^, sin and cos as operators. The simplification process partially evaluates expressions and uses is where appropriate to fully evaluate subexpressions.

B.2.13. bubble

This program sorts a list of elements using the bubble sort algorithm. To sort a list of items, type:

?- bubble_sort(Xu,Xs).

where Xu is the unsorted list and Xs is the sorted list which will be the result.

B.2.14. findall

This program implements an indirect setof in order to test the *call* predicate. Findall takes three arguments, Template, Goal, and Result. It finds all the instances of Template which satisfy Goal and returns them in Result. To use findall, type:

?- findall(Template,Goal,Result).

B.2.15. maplist

This program implements the Prolog analog to Lisp's Mapcar using *call*. Maplist takes a relation name and a list of argument lists and applies the relation to all the elements in the list. To use maplist, type:

?- maplist(R,L1,L2).

where R is the relation name, L1 is the initial list, and L2 is the result list. This program is cannot be written in Juniper-B1 as discussed below.

B.3. Juniper Translation Difficulties

A number of language and programming issues have been identified while developing these small programs. This section describes the Juniper language constructs and specific programming tasks which have proved problematic relative to the programs as they were written in Prolog.

B.3.1. setof

The set expression construct of Juniper-B1 is weaker than the standard definition of *setof/3* found in many Prolog systems[6]. In particular, the Prolog definition differs in two major aspects. First, the Prolog definition assumes a backtracking implementation. This is illustrated in the following example [6]: Consider the setof expression

setof(X, father(F,X), Kids)

which is read as "for every father who are their children". This query would produce multiple solutions in a backtracking implementation. For example, if fred is the father of ellen and brendan and john is the father of kathy and eric, then the two solutions are {X=fred, Kids=[ellen, brendan]} and {X=john, Kids=[kathy, eric]}. To express this query in Juniper-B1 requires that the template of the set expression contain all result variables and there be no other free variables. The equivalent expression in Juniper-B1 is

=(Father_Kids, {<X,Y> | father(X,Y)})

In this version, the fathers are included explicitly in the result.

There exists another interpretation of the the setof/3 expression above. Namely, the set of all children participating in the father relation regardless of the father. In Prolog, this query is distinguished via an existential variable. The above setof expression is to be contrasted with

setof(X, F^father(F,X), Kids)

which results in Kids=[ellen, brendan, kathy, eric] given the assumed database. In Juniper-B1, there is no direct equivalent. One can use the Juniper set expression which explicitly mentions all of the free variables and subsequently project a particular result variable and make a set.

B.3.2. maplist

There is no identified manner in Juniper-B1 to write the equivalent of the Prolog version of the maplist relation. This fact is attributable to the lack of a =.. operator in Juniper-B1. There does not appear to be any general translation which comes close to the usefulness of =... As =.. is used as a goal forming operation in many logic programming interpreters, it is problematic to port these kinds of programs to Juniper-B1.

B.3.3. Overloading of function names

When the system encounters a function symbol in a logic expression, it is problematic to determine whether the programmer intended the system to interpret, i.e., reduce, the expression using the function defined in the system or to leave the expression uninterpreted. Whether one decides that the default is to interpret expressions which have a defined functor symbol or the default is to leave such expressions uninterpreted, it is reasonable to expect programmers to want to use both. The symbolic differentiation program is a sample case and will be described in detail below. The remainder of this section discusses the problems in general citing examples from other systems.

In Juniper-B1, all functional expressions which use a defined function symbol will be reduced when encountered. This requires the programmer to know all function names for which there are definitions so that one can avoid using interpreted functions when appropriate. For example, if + is the defined symbol for the addition function then the functional term +(X,Y) will be reduced when encountered in some unification as per the definition in Juniper-B1. For simple cases such as arithmetic, there is a

common set set of function symbols. However, function names for list manipulation are not such a set, e.g., car, first, 1, and hd are some of the commonly used names for the function which returns the first element of a list. In general, the programming problem is one of knowing during program development if or when or which functional expressions will be reduced.

The following issues and questions arise within the Juniper framework:

- (1) If all functional expressions are to be reduced then does this preclude the use of any function symbol as a logic constant, e.g., while the logic expression $p(+ (X, Y))$ uses the function symbol $+$ in a standard way, consider the logic expression $p(+)$ in which $+$ is being used as a constant. To put this problem another way, is the function symbol $+/2$ to be considered the same function as $+/0$. In Juniper-B1, as in any language with higher order functions, this is necessarily the case. Therefore, the programmer must also be aware of all the constants used in Juniper logic programs as well, since they may refer to functions defined in the environment. As a separate observation, the identification of function symbols disregarding arity is the opposite of the logic programming case in which both predicate and function symbols implicitly refer to an associated arity.
- (2) The set of interpreted function symbols can trivially change, depending on the state of the system, e.g., version, and the state of the user program, e.g., which other functions the programmer chooses to define. This may engender obscure programming errors and limit reusability of Juniper logic programs.
- (3) If logic functional terms were actually to be indistinguishable from lists in which the head of the list were the function symbol, then the overloading problem may be exacerbated since user input, or any other arbitrary source of symbols, may cause a functional expression to be evaluable in one execution versus another. This problem has been encountered in a natural language parser written in Loglisp in which the definite article the appeared at the head of a list of tokens to be parsed. The also is a function in Loglisp. The effect was to evaluate the list of tokens which was considered incorrect by the programmer.

In summary, the programming issues of how a programmer can determine that a logic functional expression will be treated by the Juniper system, i.e., whether a particular expression is interpreted or not, is problematic. Further consideration should be given to the problem including the option of the programmer declaring evaluable function symbols.

B.3.4. The symbolic differentiation program

The symbolic differentiation program really performs two transformations on input expressions. First, it applies the standard set of differentiation rules. Second, it attempts to simplify the expression by performing what amounts to function reduction. Since this program manipulates algebraic expressions, there was a problem of representing the algebraic expressions in Juniper (see above). Also, it would seem as if the simplification step could have been done by Juniper itself. However, given that

logic variables could appear anywhere within the algebraic expressions, this was not possible.

The Prolog version of this program used the standard arithmetic operators within functional expressions. For example, here is the differentiation rule for addition:

$\text{diff}(U+V, Du+Dv) \text{ :- diff}(U, Du), \text{diff}(V, Dv).$

First, because $+$ is a defined infix operator it can appear in infix notation. Juniper-B1 ~~sv~~ ~~ax~~ does not allow infix functional expressions in logic expressions. Second, even though $+$ can be used in Prolog arithmetic expressions using *is*, the expressions above are not evaluated. Juniper-B1 would require that if $+$ is a defined function symbol then the expressions be evaluated. So, in order to translate this program to Juniper-B1 given the concerns expressed above, the program needed to be rewritten using special functors for the arithmetic operations. The function symbols in logic expressions are prefixed with *diff_*. Note, not all occurrences of a function symbol are replaced; those expressions which evaluate arithmetic expressions are still required to use the actual function symbol.

A second part of the symbolic differentiation program simplifies an algebraic formula. Simplification is performed over the operators $+$, $-$, $*$, $/$, and $.$ Expressions are simplified until no further simplification rules apply. A part of the simplification attempts to reduce expressions based on identity operations, e.g., implementing the identities $X + 0 = X$ and $X * 0 = 0$. Another part attempts to evaluate an expression when arguments to operators are coerced to integers. If Juniper (here Juniper-Bn or Juniper-C) were to reduce expressions in which logic variables appeared, then approximately half the program would be eliminated. Since Juniper-B1 does not allow logic variables to appear in functional expressions, the explicit simplification and evaluation steps remain in the program.

B.4. Acknowledgements

Many of the programs discussed in this paper are standard Prolog benchmarks or part of standard Prolog libraries. Keith Cassell translated these to the relative pure Prolog form as well as providing initial Juniper-B1 implementation. Don McKay refined the Juniper versions. Lynette Hirschman has provided general guidance and helped identify the programs to be used as well as specific uses of *setof* to investigate. Judy Pohl Clark and John Dowding provided some additional benchmarks and sample uses of *setof*. Also, this work has benefited from comments on previous versions of this paper by members of the Juniper project especially Bill Hopkins and Tom Blenko.

B.5. Summary

The series of programs described in this paper are intended to be used as an initial test set of the Juniper language. As such, they are small and simple. However, the set of programs considered do provide a diverse set of benchmarks and language problems to consider in subsequent Juniper development.

B.6. References

- [1] Tom Blenko, Preliminary SUPER-B1 Language Definition, Technical Memo, Paoli Research Center, System Development Corp., Paoli, PA, October, 1986.
- [2] Tom Blenko, Issues in the Definition of SUPER, Technical Memo, Paoli Research Center, System Development Corp., Paoli, PA, September, 1986.
- [3] John Dowding and Lynette Hirschman, MicroPundit: Version 0.1, Logic-Based System Technical Memo No. 36, Paoli Research Center, System Development Corporation, October, 1986.
- [4] Donald P. McKay, Howard Rosenshine, Keith Cassell, and David L. Matuszek, MicroKNET: Version 0.1, LBS Technical Memo No. 34, System Development Corporation, Paoli, PA, September, 1986.
- [5] Judy Pohl Clark and Donald P. McKay, Prolog Benchmark Report, Logic-Based System Technical Memo No. 35, Paoli Research Center, System Development Corporation - A Burroughs Company, September, 1986.
- [6] Leon Sterling and Ehud Shapiro, *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

B.7. Code listings

The following sections are the code listings for the benchmark programs.

B.7.1. Reversel

```
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|
|x                                     reverse.pl.juniper                      |
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|
```

DEC

```
rev([], []).
rev([H|T], L) :-
    rev(T, R),
    append(R, [H], L).
```

```
append([], L, L).
append([H|T], L, [H|U]) :-
    append(T, L, U).
```

?

B.7.2. gram2.pl

```
!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
!X                               gram2.pl.juniper
!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
!X This grammar parses a small set of English sentences.  Some sentences
!X may generate more than one parse.
```

```
!X
```

```
!X To parse a sentence, type:
```

```
!X
```

```
!X      ?- sentence(Parse_tree, Sentence, Remainder)
```

```
!X
```

```
!X where Parse_tree is the parse tree produced by Sentence, and Remainder
!X is that part of the end of the sentence that is not involved in the
!X parse.
```

```
!X
```

```
!X sample calls:
```

```
!X      ?- sentence(X, [john,was,believed,to,have,been,shot,by,mr_nose], []).
```

```
!X      ?- sentence(X, [was,john,shot], []).
```

```
!X      ?- sentence(X, [fred,likes,mary], []).
```

```
!X
```

```
connects([W|S],W,S).
```

```
sentence(SSS,S0,S) :-
```

```
    connects(S0,W,S1),
```

```
    aux_verb(W,Verb,Tense),
```

```
    noun_phrase(G_Subj,S1,S2),
```

```
    rest_sentence(q,G_Subj,Verb,Tense,SSS,S2,S).
```

```
sentence(SSS,S0,S) :-
```

```
    noun_phrase(G_Subj,S0,S1),
```

```
    connects(S1,W,S2),
```

```
    verb(W,Verb,Tense),
```

```
    rest_sentence(dcl,G_Subj,Verb,Tense,SSS,S2,S).
```

```
rest_sentence(Type,G_Subj,Verb,Tense,s(Type,L_Subj,tns(Tense1),VP),S0,S) :-
```

```
    rest_verb(Verb,Tense,Verb1,Tense1,S0,S1),
```

```
    verbtype(Verb1,VType),
```

```
    complement(VType,Verb1,G_Subj,L_Subj,VP,S1,S).
```

```
rest_verb(have,Tense,Verb,(Tense,perfect),S0,S) :-
```

```
    connects(S0,W,S),
```

```
    past_participle(W,Verb).
```

```
rest_verb(Verb,Tense,Verb,Tense,S,S).
```

```
complement(copula,be,Obj,Subj,vp(v(Verb),Obj1),S0,S):-
```

```
    connects(S0,W,S1),
```

```
    past_participle(W,Verb),
```

```
    transitive(Verb),
```



```

rest_object(Obj,Verb,Obj1,S1,S2),
agent(Subj,S2,S).

complement(transitive,Verb,Subj,Subj,vp(v(Verb),Obj1),S0,S) :-
    noun_phrase(Obj,S0,S1),
    rest_object(Obj,Verb,Obj1,S1,S).

ccplement(intransitive,Verb,Subj,Subj,vp(v(Verb)),S,S).

rest_object(Obj,Verb,SSS,S0,S) :-
    s_transitive(Verb),
    connects(S0,to,S1),
    connects(S1,Verb1,S2),
    infinitive(Verb1),
    rest_sentence(dcl,Obj,Verb1,present,SSS,S2,S).

rest_object(Obj,_,Obj,S,S).

agent(Subj,S0,S) :-
    connects(S0,by,S1),
    noun_phrase(Subj,S1,S).

agent(np(pro(someone)),S,S).

!Xnoun_phrase(np(Det,adj(Adjs),n(Noun)),S0,S) :-
noun_phrase(np(Det,n(Noun)),S0,S) :-
    connects(S0,Det,S2),
    determiner(Det),
    !X    adjectives(Adjs,S1,S2),
    connects(S2,Noun,S),
    noun(Noun).

noun_phrase(np(npr(PN)),S0,S) :-
    connects(S0,PN,S),
    proper_noun(PN).

adjectives([Adj|Adjs],S0,S) :-
    connects(S0,Adj,S1),
    adjective(Adj),
    adjectives(Adjs,S1,S).

adjectives([],S,S).

adjective(red).

aux_verb(W,V,T):- verb(W,V,T),auxiliary(V).

```

auxiliary(be).

determiner(the).

infinitive(buy).

infinitive(have).

infinitive(shoot).

infinitive(tell).

intransitive(sleep).

noun(book).

noun(cat).

noun(dog).

past_participle(been,be).

past_participle(believed,believes).

past_participle(liked,likes).

past_participle(shot,shoot).

past_participle(told,tells).

proper_noun(mr_nose).

proper_noun(fred).

proper_noun(john).

proper_noun(mary).

s_transitive(believes).

s_transitive(likes).

s_transitive(tells).

transitive(believes).

transitive(buy).

transitive(shoot).

transitive(likes).

transitive(tells).

verb(believed,believes,past).

verb(buy,buys,present).

verb(is,be,present).

verb(liked,likes,past).

verb(likes,likes,present).

verb(shot,shoot,past).

verb(told,tells,past).

verb(was,be,past).

verbtype(V,transitive):- transitive(V).

verbtype(be,copula).

?

B.7.3. peano.pl

```
!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
!X                               peano.pl.juniper
!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
!X This program does integer arithmetic and arithmetic comparisons without
!X using "is". Numbers are represented as follows:
!X     A zero is represented as 0.
!X     Positive numbers correspond to the number of "s"s to the left of 0
!X     e.g. s(0) represents 1; s(s(0)) represents 2; s(s(s(0)))
!X         represents 3, etc.
```

```
num(0).
```

```
num(s(X)):-num(X).
```

```
add(0,X,X) :- num(X).
```

```
add(s(X),Y,s(Z)):-
    add(X,Y,Z).
```

```
minus(X,Y,Z) :-
    add(Z,Y,X).
```

```
mult(X,0,0) :- num(X).
```

```
mult(X,s(Y),Z):-
    mult(X,Y,W),
    add(X,W,Z).
```

```
div(_,0,undefined).
```

```
div(X,X,s(0)) :-
    fail(equals(X,0)),
    num(X).
```

```
div(X,Y,0):- less(X,Y).
```

```
div(X,Y,Z) :-
    fail(equals(Y,0)),
    greater(X,Y),
    minus(X,Y,X1),
    div(X1,Y,Z1),
    add(s(0),Z1,Z).
```

```
mod(_,0,undefined).
```

```
mod(X,X,0):-
    fail(equals(X,0)),
    num(X).
```

```
mod(X,Y,X):- less(X,Y).
```

```
mod(X,Y,Z) :-
    fail(equals(Y,0)),
    greater(X,Y),
    minus(X,Y,X1),
    mod(X1,Y,Z).
```

```
equals(0,0).
```

```
equals(s(X),s(Y)) :-
```

```
equals(X,Y).

greater(s(X),0):-
    num(X).
greater(s(X),s(Y)):-
    greater(X,Y).

less(0,s(X)):-
    num(X).
less(s(X),s(Y)):-
    less(X,Y).

power_of(_,0,s(0)).
power_of(X,s(Y),Z):-
    power_of(X,Y,W),
    mult(X,W,Z).
```

?

B.7.4. facto.peano.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                               facto.peano.pl.juniper
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
X
X fact(N, Fact_N)
X   Fact_N is the factorial of N, e.g. fact(3,6)
X
X   This program uses Peano arithmetic, (e.g. 0 is 0, 1 is s(0), 2
X       is s(s(0)), 3 is s(s(s(0))), etc. ), so fact(3,6) is
X       actually fact( s(s(s(0))), s(s(s(s(s(s(0)))))) )
X
X Given N, the program generates Fact_N just fine and quickly fails on a
X   second attempt.
X Given Fact_N, the program generates N just fine but looks for a second
X   answer for a long time if requested to do so.
```

```
num(0).
num(s(X)):-
    num(X).
```

```
sum(X,0,X) :-
    num(X).
sum(X,s(Y),s(Z)) :-
    sum(X,Y,Z).
```

```
prod(X,0,0) :-
    num(X).
prod(X,s(Y),Z) :-
    prod(X,Y,W),
    sum(W,X,Z).
```

```
fact(0,s(0)).
fact(s(X),Z) :-
    fact(X,Y),
    prod(s(X),Y,Z).
```

?

B.7.5. fibo.peano.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%                               fibo.peano.pl.juniper
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
%
% fibo(Nth,M)
%     N is the Nth Fibonacci number, where the 0th Fibonacci number is 1
%
%     This program uses Peano arithmetic (e.g. 0 is 0, 1 is s(0), 2
%     is s(s(0)), 3 is s(s(s(0))), etc.
%
% Given Nth, the program generates M just fine and quickly fails on a second
% attempt.
% Given M, the program generaces Nth just fine but looks for a second answer
% for a long time if requested to do so.
```

```
num(0).
num(s(X)):-
    num(X).
```

```
sum(X,0,X) :-
    num(X).
sum(X,s(Y),s(Z)) :-
    sum(X,Y,Z).
```

```
fibo(0,s(0)).
fibo(s(0),s(0)).
fibo(s(s(N)),V) :-
    fibo(s(N),X),
    fibo(N,Y),
    sum(X,Y,V).
```

?

B.7.6. qsort.pl

```
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|X                                     qsort.pl
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
|X **** Quick sort ****
|X This quick sort sorts numbers in ascending order. Duplicates are not
|X removed.
```

```
|X use sort_list for a sample sort of 50 items
sort_list(Orig, Sorted) :- list50(Orig), qsort(Orig,Sorted,[]).
```

```
qsort([X|L],R,R0) :-
    partition(L,X,L1,L2),
    qsort(L2,R1,R0),
    qsort(L1,R,[X|R1]).
qsort([],R,R).
```

```
partition([X|L],Y,[X|L1],L2) :-
    <=(X, Y),
    partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
    >(X, Y),
    partition(L,Y,L1,L2).
partition([],_,[],[]).
```

```
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|X                                     SAMPLE LIST
```

```
list50( [27,74,17,33,94,18,46,83,65, 2,
        32,53,28,85,99,47,28,82, 6,11,
        55,29,39,81,90,37,10, 0,66,51,
        7,21,85,27,31,63,75, 4,95,99,
        11,28,61,74,18,92,40,53,59, 8 ]).
```

?

B.7.7. geo.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                               geo.juniper.b1
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

% [16] **** Database Manipulations ****

% This database is created by J. A. Robinson et al.

% population(afghanistan,20900000). ==> 161 clauses.

% adjoins(canada,usa). ==> 313 clauses.

% open_water(atlantic_ocean). ==> 40 clauses.

% country(moscow,ussr). ==>220 clauses.

% region(canada,north_america). ==> 162 clauses.

% produces(ussr,oil,491,1975). ==> 252 clauses.

% belongs(canada,nato). ==> 177 clauses.

%

% Below are some sample queries to use on the database.

%

% What is a city in Japan?

db1(S):- country(S, japan).

% Which far-east country is landlocked?

db3(C) :- region(C,far_east), iscountry(C), landlocked(C).

% List all the far-east countries which are landlocked.

db4({C | db3(C)}).

% Which country borders two seas?

db5(C) :- iscountry(C), S = {X | border_sea(C, X)}, size(S,2).

border_sea(C,X) :- borders(C,X), open_water(X).

size([_,_],2).

% Which countries border two seas?

db6({C | db5(C)}).

% Which country bordering the Mediterranean borders a country that is

% bordered by a country whose population exceeds the population of the USA?

db7(C) :-

population(usa,Y), borders(C,mediterranean_sea), iscountry(C),

borders(C,C1), iscountry(C1), borders(C1,C2),

population(C2,X), X>Y.

% Which countries bordering the Mediterranean border a country that is

% bordered by a country whose population exceeds the population of the USA?

db8({C | db7(C)}).

% What is the oil production of countries which are members of OPEC but

% not the Arab League?


```
db9([C,P] | oil_production(C,P))).
```

```
oil_production(C,P) :-
```

```
  belongs(C,opeac), fail(belongs(C,arab_league)), produces(C,oil,P,1975).
```

```
% Which is the ocean that borders African countries and that  
% borders European countries?
```

```
db10(X) :-
```

```
  open_water(X), borders(X,C), region(C,africa), iscountry(C),
```

```
  borders(X,C1), region(C1,europe), iscountry(C1).
```

```
% Which countries belong to the organization Org?
```

```
db11([C | belongs(C,Org)]).
```

```
% What are the organizations to which a country may belong and which countries  
% belong to each?
```

```
db12([Org,Countries] | Countries = {C | belongs(C,Org)}).
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
%
```

```
% Database elided.....
```

```
?
```

B.7.8. foxes.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                               foxes.pl.juniper
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
/*****
/*      The Farmer and the River Problem      */
/*****
```

```
/*****
/* Problem:
```

The Farmer has a goose, grain, and a fox. He needs to cross a river, but his boat will only hold himself and one other article. The problem arises because 1) the grain and the goose cannot be left unattended, nor 2) can the goose and the fox.

The problem solver keeps track of the current state as a predicate of the form:

state(Farmer,Fox,Goose,Grain).

Where Farmer is abbreviated F, Fox is X, Goose is G and Grain is A. Each place can take on a value of either 1 or 2, each number representing a side of the river.

*/

```
/* This is the top level predicate whose purpose is to establish
whether there is a non-circular path from the Start state to the
Goal state. Solution is that path. path/3 calls path/4 with a list
of the Start state as the germ of the already seen states list. To
generate a solution, type:
```

?- path(state(1,1,1,1), state(2,2,2,2), Solution).

There are two solutions to this query.

*/

```
path(Start,Goal,Solution) :-
    path(Start,[Start],Goal,Solution).
```

```
/* A path from State to Goal exists if State is linked with Goal.
Path is the list of states encountered up to now. Solution is Path
prepended with Goal.
```

*/

```
path(State,Path,Goal,[Goal|Path]) :-
    linked(State,Goal).
```

```
/* A path from State1 to Goal exists if there exists a State2 which
has not yet been encountered, which is linked with State1 and for
```

which a path exists to Goal. State2 is part of the list of states encountered up to now and Solution is the list of states leading from the original state to Goal.

*/

```
path(State1,Path,Goal,Solution) :-
    linked(State1,State2),
    fail(member(State2,Path)),
    path(State2,[State2|Path],Goal,Solution).
```

/* There are four cases for linked states:

Only the farmer changes sides */

```
linked(state(F1,X,G,A),state(F2,X,G,A)) :-
    opposite(F1,F2),
    legal(state(F2,X,G,A)).
```

/* The farmer and the fox change sides */

```
linked(state(F1,F1,G,A),state(F2,F2,G,A)) :-
    opposite(F1,F2),
    legal(state(F2,F2,G,A)).
```

/* The farmer and the goose change sides */

```
linked(state(F1,X,F1,A),state(F2,X,F2,A)) :-
    opposite(F1,F2),
    legal(state(F2,X,F2,A)).
```

/* The farmer and the grain change sides */

```
linked(state(F1,X,G,F1),state(F2,X,G,F2)) :-
    opposite(F1,F2),
    legal(state(F2,X,G,F2)).
```

/* A legal state is one where either the farmer is on the same side as the goose or is on the same side as both the fox and the grain. In the latter case, the condition that the farmer be on the opposite side from the goose is necessary to prevent resolutions that repeat states already covered by the first legal/1 definition.

*/

```
legal(state(F,_,F,_)).
legal(state(F,F,G,F)) :- opposite(F,G).
```

/* Test for membership of an element in a list */

```
member(E,[E|_]).
member(E,[_|T]) :- member(E,T).
```

/* Define sides 1 and 2 as being opposite */

```
opposite(1,2).
opposite(2,1).
```

?

B.7.9. sets.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                                     sets.pl
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

% The following predicates define various operations on sets. Sets
% are represented as an unordered list of unique elements. The
% operations given are: member, subset, union, intersection, subtract,
% and seteq.

DEC

% member(Member,Set) is true if Member is a member of Set. It works
% with the following modes: (+,+), (+,-), and (-,+).

```
member(X,[X1|Y]) :- X = X1 -> true;member(X,Y).
```

% subset(Set1,Set2) is true if Set1 is a subset of Set2. It works with
% the following modes: (+,+) and (+,-).

```
subset([],_Set2).
subset([X|Set1],Set2) :- member(X,Set2),subset(Set1,Set2).
```

% union(Set1,Set2,Union) is true if the union of Set1 and Set2 is Union.
% It works with the following modes: (+,+,+), (+,+,-), (-,-,+), and (+,-,+).

```
union([],Y,Y).
union([X|Xs],Ys,Zs) :- member(X,Ys),fail(member(X,Xs)),union(Xs,Ys,Zs).
union([X|Xs],Ys,[X|Zs]) :- union(Xs,Ys,Zs),fail(member(X,Ys)).
```

% intersection(Set1,Set2,Intersection) is true if the intersection of
% sets 1 and 2 is Intersection. It works in the following modes:
% (+,+,+), (+,+,-), and (-,-,+).

```
intersection([],_X,[]).
intersection([X|Xs],Ys,[X|Zs]) :- member(X,Ys),intersection(Xs,Ys,Zs).
intersection([X|Xs],Ys,Zs) :- fail(member(X,Ys)),intersection(Xs,Ys,Zs).
```

% subtract(Set1,Set2,Result) is true if Set1 - Set2 = Result.
% It works in the following modes: (+,+,+) and (+,+,-).

```
subtract([],_Y,[]).
subtract([X|Xs],Ys,Zs) :- member(X,Ys),subtract(Xs,Ys,Zs).
subtract([X|Xs],Ys,[X|Zs]) :- fail(member(X,Ys)),subtract(Xs,Ys,Zs).
```

% seteq(Set1,Set2) is true if Set1 contains the same elements as Set2.
% It works in the following modes: (+,+) and (+,-).

seteq(X,Y) :- subset(X,Y),subset(Y,X).

?

B.7.10. fibo.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                               fibo.pl.juniper                        X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
fibo(0,1).
fibo(1,1).
fibo(In, Out1 + Out2) :-
    >(In, 1),
    fibo(In - 1, Out1),
    fibo(In - 2, Out2),
fibo(Neg, 'ERROR - negative number as input') :-
    <(Neg, 0).
```

?

B.7.11. queens.pl

```
X-----
X Queens on a chess board problem...
X Only two solution on a 4x4 board...

X get_int returns a positive integer that is =< Max

get_int(1,1).
get_int(Max, Int) :-
    Max > 1,
    Int = Max.
get_int(Max, Int) :-
    Max > 1,
    Max1 is Max - 1,
    get_int(Max1, Int).

queens(Board_size, Soln) :- solve(Board_size, [], Soln).

X      solve accumulates the positions of occupied squares

solve(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]).
solve(Board_size, Initial, Final) :-
    newsquare(Board_size, Initial, Next),
    solve(Board_size, [Next|Initial], Final).

X      newsquare generates legal positions for next queen

newsquare(Board_size, [], square(1, X)) :-
    get_int(Board_size, X).
newsquare(Board_size, [square(I, J) | Rest], square(X, Y)) :-
    X is I + 1,
    get_int(Board_size, Y),
    +(threatened(I, J, X, Y)),
    safe(X, Y, Rest).

X      safe checks whether square(X, Y) is threatened by any
X      existing queens

safe(_, _, []).
safe(X, Y, [square(I, J) | L]) :-
    +(threatened(I, J, X, Y)),
    safe(X, Y, L).

X      threatened checks whether squares (I, J) and (X, Y)
X      threaten each other

threatened(I, _, X, _) :-
    (I = X).
threatened(_, J, _, Y) :-
    (J = Y).
threatened(I, J, X, Y) :-
    (U is I - J),
    (V is X - Y),
```

```
(U = V).  
threatened(I, J, X, Y) :-  
    (U is I + J),  
    (V is X + Y),  
    (U = V).
```


B.7.12. queens.pl

```
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|X                                     queens.pl.juniper
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DEC

|X-----
|X Queens on a chess board problem...
|X Only two solution on a 4x4 board...

|X get_int returns a positive integer that is <= Max

get_int(1,1).
get_int(Max, Max) :-
    >(Max,1).
get_int(Max, Int) :-
    >(Max,1),
    get_int(Max - 1, Int).

queens(Board_size, Soln) :- solve(Board_size, [], Soln).

|X      solve accumulates the positions of occupied squares

solve(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]).
solve(Board_size, Initial, Final) :-
    newsquare(Board_size, Initial, Next),
    solve(Board_size, [Next|Initial], Final).

|X      newsquare generates legal positions for next queen

newsquare(Board_size, [], square(1, X)) :-
    get_int(Board_size, X).
newsquare(Board_size, [square(I, J) | Rest], square(X, Y)) :-
    =(X, I + 1),
    get_int(Board_size, Y),
    fail(threatened(I, J, X, Y)),
    safe(X, Y, Rest).

|X      safe checks whether square(X, Y) is threatened by any
|X      existing queens
|X
safe(_, _, []).
safe(X, Y, [square(I, J) | L]) :-
    fail(threatened(I, J, X, Y)),
    safe(X, Y, L).

|X      threatened checks whether squares (I, J) and (X, Y)
|X      threaten each other

threatened(I, _, I, _).
threatened(_, J, _, J).
threatened(I, J, X, Y) :-
```

```
      =(I - J, X - Y).  
threatened(I, J, X, Y) :-  
      =(I + J, X + Y).
```

?

B.7.13. diff.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%
%                               DIFF.PL
%
% adapted from a program written by Tim Finin
%
% REQUIRED PREDs: is, integer
%      "integer" can be easily written using "is"
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

ddx(E1,E2) :-
    diff(E1,E),
    simplify(E,E2).

diff(U+V,Du+Dv) :-
    diff(U,Du),
    diff(V,Dv).

diff(U-V,Du-Dv) :-
    diff(U,Du),
    diff(V,Dv).

diff(U*V,Du*V + U*Dv) :-
    diff(U,Du),
    diff(V,Dv).

diff(U^N,N*U^(N-1)*Du) :-
    integer(N).
    N1 is N-1,
    diff(U,Du).

diff(-U,-Du) :- diff(U,Du).

diff(U/V,(V*Du+U*Dv)/V*V) :-
    diff(U,Du),
    diff(V,Dv).

diff(sin(U),cos(U)*Du) :- diff(U,Du).

diff(cos(U),-sin(U)*Du) :- diff(U,Du).

diff(x,1).

diff(C,0) :-
    +(C = U+V),
    +(C = U-V),
    +(C = U*V),
    +(C = U^V),
    +(C = -U),
    +(C = U/V),
    +(C = sin(U)),
    +(C = cos(U)),
```

```

+(X == C).

/*
Simplification of arithmetic Exeressions --
simplify(E1,E2) is true if E2
is a simplified version of E1
*/

simplify(X,Y) :- s(X,Y).

% try to recursively simplify args

s(X+Y,E2) :- s(X,X2), s(Y,Y2), s1(X2+Y2,E2).
s(X-Y,E2) :- s(X,X2), s(Y,Y2), s1(X2-Y2,E2).
s(X*Y,E2) :- s(X,X2), s(Y,Y2), s1(X2*Y2,E2).
s(X/Y,E2) :- s(X,X2), s(Y,Y2), s1(X2/Y2,E2).
s(X^Y,E2) :- s(X,X2), s(Y,Y2), s1(X2^Y2,E2).
s(E1,E2) :- s1(E1,E2).

% simplify1 keeps trying simp rules until none applies

s1(E1,E2) :- simp(E1,Es), !, s1(Es,E2).
s1(E1,E1).

% simp tries applying a simple simplification rule

simp(X+0,X).
simp(0+X,X).

simp(X-0,X).
simp(0-X,-X).

simp(-(-X),X).

simp(1+X,X).
simp(X+1,X).
simp(_+0,0).
simp(0+_,0).
simp(A^X1+A^X2, A^Sum) :- s(X1+X2,Sum).

simp(X/1,X).
simp(0/_ ,0).
simp(X/X,1).
simp(X/(X^A),1/X^Diff) :- s(A-1,Diff).
simp(X^A/(X^B),X^Diff) :- s(A-B,Diff).

simp(X^1,X).
simp(_^0,1).

% if all args are numbers, evaluate

simp(X+Y,N) :-
    integer(X),

```

integer(Y),
N is X+Y.

simp(X-Y,N) :-
integer(X),
integer(Y),
N is X-Y.

simp(X*Y,N) :-
integer(X),
integer(Y),
N is X*Y.

simp(X/Y,N) :-
integer(X),
integer(Y),
N is X/Y.

simp(X^Y,N) :-
integer(X),
integer(Y),
N is X^Y.

simp(-X,N) :-
integer(X),
N is 0-X.

X -----

simp(A*(B+C),X=C) :-
integer(A),
integer(B),
X is A * B.

simp(A*(B+C),X=B) :-
integer(A),
integer(C),
X is A * C.

simp(A*(B+C),X=A) :-
integer(B),
integer(C),
X is B * C .

B.7.14. diff.pl

```

|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|%                               diff.pl.juniper
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

DEC

```

ddx(E1,E2) :-
    diff(E1,E),
    simplify(E,E2).

```

```

diff(diff_+(U, V),diff_+(Du, Dv) :-
    diff(U,Du),
    diff(V,Dv).

```

```

diff(diff_-(U, V),diff_-(Du, Dv)) :-
    diff(U,Du),
    diff(V,Dv).

```

```

diff(diff_*(U, V),diff_*(diff_+(Du, V), diff_*(U, Dv)) :-
    diff(U,Du),
    diff(V,Dv).

```

```

diff(diff_^(U, N),diff_*(diff_*(N, diff_^(U, N1)), Du)) :-
    integer(N),
    N1 is N-1,
    diff(U,Du).

```

```

diff(diff_-(U),diff_-(Du)) :- diff(U,Du).

```

```

diff(diff_/(U, V),
    (diff_/(diff_+(diff_*(V, Du), diff_*(U, Dv))), diff_*(V, V)) :-
    diff(U,Du),
    diff(V,Dv).

```

```

diff(diff_sin(U),diff_*(diff_cos(U), Du)) :- diff(U,Du).

```

```

diff(diff_cos(U),diff_*(diff_-(diff_sin(U)), Du)) :- diff(U,Du).

```

```

diff(x,1).

```

```

diff(C,0) :-
    fail(C = diff_+(U, V)),
    fail(C = diff_-(U, V)),
    fail(C = diff_*(U, V)),
    fail(C = diff_^(U, V)),
    fail(C = diff_-(U)),
    fail(C = diff_/(U, V)),
    fail(C = diff_sin(U)),
    fail(C = diff_cos(U)),
    . fail(x == C).

```

/*

```

Simplification of arithmetic Exeressions --
simplify(E1,E2) is true if E2
is a simplified version of E1
*/

```

```

simplify(X,Y) :- s(X,Y).

```

```

!X try to recursively simplify args

```

```

s(diff_+(X, Y),E2) :- s(X,X2), s(Y,Y2), s1(diff_+(X2, Y2),E2).
s(diff_-(X, Y),E2) :- s(X,X2), s(Y,Y2), s1(diff_-(X2, Y2),E2).
s(diff_*(X, Y),E2) :- s(X,X2), s(Y,Y2), s1(diff_*(X2, Y2),E2).
s(diff_/(X, Y),E2) :- s(X,X2), s(Y,Y2), s1(diff_/(X2, Y2),E2).
s(diff_^(X, Y),E2) :- s(X,X2), s(Y,Y2), s1(diff_^(X2, Y2),E2).
s(E1,E2) :- s1(E1,E2).

```

```

!X simplify! keeps trying simp rules until none applies

```

```

s1(E1,E2) :- simp(E1,E3), !, s1(E3,E2).
s1(E1,E1).

```

```

!X simp tries applying a simple simplification rule

```

```

simp(diff_+(X, Y), M) :-
    diff_plus(X, Y, M).

```

```

simp(diff_-(X), M) :-
    diff_minus(X, M).

```

```

simp(diff_-(X, Y), M) :-
    diff_difference(X, Y, M).

```

```

simp(diff_*(X, Y), M) :-
    diff_times(X, Y, M).

```

```

simp(diff_/(X, Y), M) :-
    diff_divide(X, Y, M).

```

```

simp(diff_^(X, Y), M) :- diff_exp(X, Y, M).

```

```

!X -----

```

```

diff_plus(X, 0, X).
diff_plus(0, X, X).
diff_plus(X, Y, M) :-
    integer(X),
    integer(Y),
    =(M, +(X, Y)).

```

```

diff_difference(X, 0, X).
diff_difference(0, X, diff_-(X)).
diff_difference(X, Y, M) :-
    integer(X),
    integer(Y),

```

```

=(N, -(X,Y)).

diff_minus(diff_-(diff_-(X)), X).
diff_minus(X, N) :-
    integer(X),
    =(N, -(X)).

diff_times(_, 0, 0).
diff_times(0, _, 0).
diff_times(X, 1, X).
diff_times(1, X, X).
diff_times(diff_^(A, X1), diff_^(A, X2), diff_^(A, Sum)) :-
    s(diff_+(X1, X2), Sum).
diff_times(X, Y, N) :-
    integer(X),
    integer(Y),
    =(N, *(X,Y)).
diff_times(A, diff_+(B, C), diff_+(N, C)) :-
    integer(A),
    integer(B),
    =(N, *(A, B)).
diff_times(A, diff_+(B, C), diff_+(N, B)) :-
    integer(A),
    integer(C),
    =(N, *(A, C)).
diff_times(A, diff_+(B, C), diff_+(N, A)) :-
    integer(B),
    integer(C),
    =(N, *(B, C)).

diff_divide(0, _, 0).
diff_divide(X, 1, X).
diff_divide(X, X, 1).
diff_divide(X, (diff_^(X, A)), diff_/((1, diff_^(X, Diff)) :-
    s(diff_-(A, 1), Diff).
diff_divide(diff_^(X, A), diff_^(X, B), diff_^(X, Diff)) :-
    s(diff_-(A, B), Diff)
diff_divide(X, Y, N) :-
    integer(X),
    integer(Y),
    =(N, /(X,Y)).

diff_exp(X, 1, X).
diff_exp(_, 0, 1).
diff_exp(X, Y, Z) :-
    integer(X),
    integer(Y),
    =(N, ^(X, Y)).

```


?

B.7.15. bubble.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%                               bubble.pl
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
% **** Bubble sort ****
% This bubble sort sorts numbers in ascending order. Duplicates are not
% removed.
```

```
% use sort_list for a sample sort of 50 items
```

```
sort_list(Orig, Sorted) :- list50(Orig), bubble_sort(Orig,Sorted).
```

```
bubble_sort(Unsorted,Sorted) :-
    append(Dummy,[X,Y|Rest],Unsorted),
    X > Y,
    append(Dummy,[Y,X|Rest],Temp),
    bubble_sort(Temp,Sorted).
bubble_sort(S,S).
```

```
append([],L,L).
append([X|L1],L2,[X|L3]) :-
    append(L1,L2,L3).
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%                               SAMPLE LIST
```

```
list50( [27,74,17,33,94,18,46,83,65, 2,
        32,53,28,85,99,47,28,82, 6,11,
        55,29,39,81,90,37,10, 0,66,51,
        7,21,85,27,31,63,75, 4,95,99,
        11,28,61,74,18,92,40,53,59, 8 ]).
```

B.7.16. bubble.pl

```
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|%                               bubble.pl
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

DEC

|% **** Bubble sort ****
|% This bubble sort sorts numbers in ascending order.  Duplicates are not
|% removed.

|% use sort_list for a sample sort of 50 items

sort_list(Orig, Sorted) :- list50(Orig), bubble_sort(Orig,Sorted).

bubble_sort(Unsorted,Sorted) :-
    append(Dummy,[X,Y|Rest],Unsorted),
    <(X,Y),
    append(Dummy,[Y,X|Rest],Temp),
    bubble_sort(Temp,Sorted).
bubble_sort(S,S).

append([],L,L).
append([X|L1],L2,[X|L3]) :-
    append(L1,L2,L3).

|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|%                               SAMPLE LIST
|

list50( [27,74,17,33,94,18,46,83,65, 2,
        32,53,28,85,99,47,28,82, 6,11,
        55,29,39,81,90,37,10, 0,66,51,
        7,21,85,27,31,63,75, 4,95,99,
        11,28,61,74,18,92,40,53,59, 8 ] ).
```

?

B.7.17. maplist.pl

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%
%               call.pl
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

% **** MapList ***
% This program takes a function name as the first argument and a list as the
% second argument and applies the function to the elements of the list. The
% results are returned in the third argument as a list.

maplist(_,[],[]).
maplist(Func,[X|Rest],[Y|Result]) :-
    Goal =.. [Func,X,Y],
    call(Goal),
    maplist(Func,Rest,Result).

plus1(X,Y) :- Y is X + 1.

test(X,X).
```

B.7.18. maplist.pl

```
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|
|X                                     call.pl                                |
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|
```

DEC

```
|X **** MapList ****
|X This program takes a function name as the first argument and a list as the
|X second argument and applies the function to the elements of the list. The
|X results are returned in the third argument as a list.
```

```
maplist(_,[],[ ]).
maplist(Func,[X|Rest],[Y|Result]) :-
    Goal =.. [Func,X,Y],
    call(Goal),
    maplist(Func,Rest,Result).
```

```
plus1(X,X + 1).
```

```
test(X,X).
```

?

B.7.19.

findall.pl

```
!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
!X                               findall.pl
!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

DEC

```
!X **** Findall ****
```

```
!X This program finds all the instances of Template which satisfy Goal and
!X returns them in Result.
```

```
findall(Template,Goal,Result) :-
    =(Result, {Template | call(Goal)}).
```

?

B.7.20. findall.pl

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%
%               findall.pl
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

% **** Findall ****
% This program finds all the instances of Template which satisfy Goal and
% returns them in Result.

findall(Template,Goal,Result) :-
    setof(Template,call(Goal),Result).
```

Appendix C
Juniper-B1 Unification

by
Tom M. Blenko

Table of Contents

Section	Title	Page
C.1	Martelli and Montanari revisited	C-1
C.1.1	Preliminaries	C-1
C.1.2	Transformation rules	C-2
C.1.3	The algorithm	C-3
C.1.4	Properties of the algorithm	C-4
C.2	Juniper-B1 unification algorithm	C-5
C.2.1	Juniper-B1 preliminaries	C-6
C.2.2	Reduction to type	C-6
C.2.3	Unifiers and satisfiability	C-7
C.2.4	Most general unifiers	C-8
C.2.5	Juniper-B1 transformation rules	C-12
C.2.6	Juniper-B1 unification algorithm	C-12
C.2.7	Properties of Juniper-B1 unification	C-14
C.2.8	Juniper-B1 unification and concurrent reduction semantics	C-19
	References	C-20
Appendices		
C-A	Proof of theorem 5	C-20

Appendix C

Juniper-B1 Unification

Our approach to characterizing the Juniper-B1 unification algorithm is to extend the model for unification provided by Martelli and Montanari [1]. We provide a brief review of their model, followed by the extensions required for the Juniper-B1 unification algorithm, and characterization of its computational properties.

C.1. Martelli and Montanari revisited

C.1.1. Preliminaries

A *term* is defined as follows:

1. A variable is a term;
2. A constant symbol is a term;
3. If $t_1, \dots, t_n$ are terms and f is an n -ary function symbol, then $f(t_1, \dots, t_n)$ is a term.

A *substitution* θ is a mapping from terms to terms. It is the identity function almost everywhere, except for at a finite number of variables. The substitution makes no changes other than to replace the variables $x_1, \dots, x_n$ with the terms $t_1, \dots, t_n$ is signified by $\theta = \{ t_1/x_1, \dots, t_n/x_n \}$.

A *unification problem* may be expressed as an equation

$$t = t'$$

or as a set of such equations.

A *solution*, or *unifier*, of an equation is any substitution which makes the two terms equal, i.e., $\theta(t) = \theta(t')$. This definition is readily extended to unifiers over sets of equations, which is the form in which our unification problems are most often expressed.

A *most general unifier* θ is a unifier such that if θ and any other substitution θ' are unifiers for a set of equations, then there exists a substitution σ such that $\theta' = \sigma.\theta$ ('.' denotes functional composition).

A set of equations is in *solved form* if and only if it satisfies the following conditions:

1. The equations are in the form

$$x_i = t_i, \quad i = 1, \dots, k$$

where x_i is a variable and t_i is a term.

2. Every variable which is the left-hand side of an equation occurs only there (note that satisfaction of this condition subsumes the occurs check).

The substitution $\theta = \{ t_1/x_1, \dots, t_k/x_k \}$ is a solution for a set of equations in solved form and is (trivially) its most general unifier.

C.1.2. Transformation rules

Two transformation rules are used as part of the unification algorithm:

T1 (*Term decomposition*): For any equation of the form

$$f'(t_1, \dots, t_n) = f''(t'_1, \dots, t'_n)$$

where f' and f'' are n -ary function symbols and f' and f'' are equal, replace the equation with the set of equations

$$\begin{aligned} t_1 &= t'_1 \\ t_2 &= t'_2 \\ &\vdots \\ t_n &= t'_n \end{aligned}$$

In the special case that $n=0$, the equation is simply erased. Note that this special case encompasses equality of constants.

T2 (Variable elimination): For any equation of the form

$$z = t$$

where z is a variable and t is a term, a new set of equations is obtained by applying the substitution $\theta = \{ t/z \}$ to all equations in the current set without erasing $z = t$.

C.1.3. The algorithm

Martelli and Montanari define the unification algorithm in terms of four solution-preserving reduction rules on sets of equations:

R1 (Term reduction): For any equation of the form

$$f'(t_1, \dots, t_n) = f''(t_1, \dots, t_n)$$

where f' and f'' are n -ary function symbols, if f' is not equal to f'' , halt with failure; otherwise apply the term decomposition transformation T1.

R2 (Variable elimination): For any equation of the form

$$z = t$$

where z is a variable and t is a term, if z occurs in t then halt with failure; otherwise if z occurs in another equation, apply the variable elimination transformation T2.

R3 (Left ordering): For any equation of the form

$$t = z$$

where z is a variable and t is not, erase the equation and replace it with

$$z = t$$

R4 (Variable identity elimination): For any equation of the form

$$x = x$$

where x is a variable, erase the equation.

The unification algorithm, given an initial set of equations, consists of selecting equations and applying reduction rules in an arbitrary order until no reduction rule is applicable to any equation, or until the algorithm halts with failure.

C.1.4. Properties of the algorithm

Two principal results are proven for this algorithm:

- (i) The algorithm terminates irrespective of the order in which equations and reduction rules are selected.
- (ii) If the algorithm terminates with failure, the set of equations has no unifier. If it terminates with success, the resultant set of equations is in solved form, and (trivially) has a most general unifier.

- (i) is shown by defining an integer metric on sets of equations that is positive and monotonically decreasing under application of the given reduction rules.
- (ii) is demonstrated from the following observations:

The algorithm halts with failure if R1 encounters an inequality, or if R2 encounters a violation of the occurs check, and in either case there is no unifier.

For the case that the algorithm terminates with success, it is shown by proving two theorems that T1 and T2 (and hence R1 and R2) do not change the unifiers of the set of equations, and the same is trivially true for R3 and R4. Therefore the unifiers of the resultant set of equations are exactly the same as the unifiers of the initial set of equations for each step in the unification procedure.

When the algorithm halts with success, inapplicability of R1 and R3 indicates that all equations are in the form $x=t$, and inapplicability of R2 and R4 indicates that each variable x that appears on the left-hand side of an equation appears only there.

Hence the resulting set of equations is in solved form and defines the most general

unifier.

C.2. Juniper-B1 unification algorithm

Unification in Juniper-B1 is more powerful than, and properly subsumes, the unification algorithm just described. This is primarily due to the introduction of *function applications*, terms that syntactically resemble the first-order terms defined above. Function applications have the property that they may be rewritten to other terms in accordance with a *functional reduction policy*. The rewriting of terms can also be viewed as a means for extending equality, so that a function application term is treated by the unification algorithm as equivalent to the value to which it can be functionally reduced.

Juniper-B1 unification can be defined informally as follows:

1. The unification algorithm matches two terms in the same sense that Martelli and Montanari's algorithm does. It determines whether a substitution exists that satisfies all equations in the unification problem, and what a member of the class of most general such substitutions is (see following).
2. The algorithm is extended to encompass reduction of function applications. Functional reduction is applied only when required to perform matching, and is applied according to the Juniper-B1 functional reduction policy, termed *reduction to type*, which is discussed below. The precise conditions under which functional reduction is applied are defined in the algorithm below.
3. Functional reduction introduces the possibility of non-termination; non-termination will in all cases be regarded as failure to unify.
4. Higher-order functions (function applications that produce functions as their values) are prohibited.
5. Functional reduction may be applied only to fully grounded function applications, i.e., those containing no variables. It is the programmer's responsibility to see that there exists an ordering on unification of subterms under which a function application subterm will be grounded before it needs to be reduced as part of the unification (otherwise the unification problem and the program are illegal).
6. The unification algorithm for Juniper-B1, as defined below, has the Church-Rosser property: the unification algorithm computes a member of the set of most general unifiers, if the set is non-empty, irrespective of the order in which the reduction rules are applied.

The purpose of the sections that follow is to revise the Martelli and Montanari model to reflect Juniper-B1's extended unification algorithm, and to establish the associated correctness and termination properties.

C.2.1. Juniper-B1 preliminaries

Juniper-B1 terms differ from the those of the first-order term systems addressed by Martelli and Montanari. The principal difference is that Juniper-B1 contains both *constructions*, which are identical to the irreducible, record-like terms found in the first-order term systems, and *function applications*, which are treated as functionally-reducible expressions (e.g., the function named "merge" can be applied to two arguments which are expressions denoting sorted lists, and the reduced value of this function application is an expression denoting the merged list).

Consequently, in Juniper-B1

1. Variables are terms;
2. If $t_1, \dots, t_n$ are terms and f is an n -ary constructor, $n \geq 0$, then $f(t_1, \dots, t_n)$ is a term called a construction;
3. If $t_1, \dots, t_n$ are terms and f is an n -ary reducible function, $n \geq 0$, then $f(t_1, \dots, t_n)$ is a term called a function application.

The definitions of *substitution*, *unification problem*, and *solved form* are extended to incorporate the expanded term system in the obvious way.

Defining *unifier* and *most general unifier* is more difficult, and requires a careful treatment of the role functional reduction plays in Juniper-B1's unification algorithm.

C.2.2. Reduction to type

The Juniper-B1 functional reduction policy consists of a form of lazy evaluation termed *reduction to type*. For an expression denoting a constant, the reduced form of the expression is the constant as one would expect. For an expression which denotes a list or some other constructor term, however, the reduced form is the constructor, but with arguments that may or may not be reduced. For example, the reduction-to-type value of the merge application mentioned above is a list-constructor term whose first argument has as its value the first element of the merged list and whose second argument is a not-necessarily-evaluated merge of the remaining lists. If the first element of the first list is less than the first element of the second list, the functional reduction would look like this:

$\text{merge}(\text{list1}, \text{list2}) \ll \text{reduces to} \gg \text{cons}(\text{head}(\text{list1}), \text{merge}(\text{tail}(\text{list1}), \text{list2}))$

Reduction to type permits one, for example, to access the second element in a list (by reducing the list expression to type, reducing the tail of the list expression to type, and then accessing the head of that tail) without having to compute the value of the first element.

The intended consequence of this functional reduction policy for the Juniper-B1 unification algorithm is that functional reduction will be performed only in those cases when it is necessary to determine unifiability. Since reduction to type does not necessarily terminate, the laziness of the functional reduction policy allows the unification algorithm to avoid non-termination in cases when this is possible. The algorithm provided below identifies, in the form of a single reduction rule, precisely those cases in which reduction is necessary.

C.2.3. Unifiers and satisfiability

In Martelli and Montanari's algorithm, a unifier was a substitution that mapped a unification problem to a set of equalities. That is to say, the original equations, under the unifying substitution, maps to a set of equations that are said to be *satisfied*. For Juniper-B1, we start from this same notion, but need to provide a more precise definition of what it means for a set of equations to be satisfied.

In general, we say that an equation is satisfied if the term on its left-hand side and the term on its right-hand side are equal. The Juniper-B1 definition of equality addresses equality over function application terms using this rule: a term t is equal to a reducible term t' if and only if it is equal to its reduced value.

In order to characterize satisfaction of a unification problem by a presumptive unifier, this definition of equality is not strong enough. For example, in the unification problem

$$x = f(1)$$

where x is a variable, the substitution $\theta = \{ f(1)/x \}$ is the presumptive solution consistent with Juniper-B1's lazy form of unification, as outlined above (it is the one found without performing unnecessary functional reductions). However, satisfaction of this unification problem, under the Juniper-B1 definition of equality, would be undecidable in the case that reduction to type of $f(1)$ did not terminate. What is necessary for determining satisfiability, then, is a stronger form of equality in which equality of two terms may be determined via syntactic identity without computing their reduced values.

Note that the assertion is not that, given a unification problem consisting of

$$f(1) = f(1)$$

the unification procedure would terminate if reduction to type of $f(1)$ did not terminate (it would not), but that $\theta = \{ f(1)/z \}$ is a permissible solution to the earlier unification problem, even though showing that it is a solution would be non-terminating under the weak definition of equality found in Juniper-B1.

A solution, or *unifier*, then, is defined as any substitution which makes the left- and right-hand sides of a (single-equation) unification problem equal under the stronger equality relation. The multiple-equation definition of a solution or unifier follows directly.

C.2.4. Most general unifiers

The problem of defining a most general unifier is more complex. We first outline some of the issues pertaining to what we might expect a most general unifier to be, and then present the definition chosen for the Juniper-B1 unification algorithm.

The problem

The definition of the most general unifier for a unification problem within the Martelli and Montanari model is based upon a simple, intuitive ordering among unifying substitutions: a substitution θ is more general than a substitution θ' if and only if for every term t , $\theta(t)$ is no more instantiated than $\theta'(t)$. Extending this notion to function application terms in Juniper-B1 is straightforward.

In Juniper-B1, it is tempting to insert issues related to the laziness of functional reduction into the generality ordering as well. In particular, should not the property of laziness somehow be part of the generality ordering, so that binding of a variable to an unreduced term is treated as more general in the ordering than binding to the reduced value of the term? Interestingly enough, our answer is "no", for the following reasons:

(i) *Church-Rosser property*. It is a desideratum from both the programmer's and the implementor's points of view that Juniper-B1 unification have the Church-Rosser property. If, for example, the term $f(1)$ is functionally reducible to the value δ , then consider the unification problem

$$\begin{aligned} x &= f(1) \\ x &= \delta \end{aligned}$$

Any algorithm remotely like Martelli and Montanari's would call for binding of z using one equation and substitution for z in the other. However, if the substitution $\theta = \{ f(1)/z \}$ were taken to be more general than $\theta' = \{ \delta/z \}$, the algorithm

would have to examine the whole unification problem to determine which equation to select in order to produce the most general binding for x . This would clearly be a negative factor with respect to efficiency of the algorithm, but more important, it would violate the Church-Rosser property, that the equations may be selected and the reduction rules applied in an arbitrary order.

(ii) *Non-existence of unique most general unifiers.* The previous problem would be further complicated if the unification problem was extended to

$$\begin{aligned}x &= f(1) \\x &= 6 \\x &= g(2)\end{aligned}$$

where $g(2)$ is a function application with a reduced value of 6. In this case, there is no criterion for determining which of $f(1)$ and $g(2)$ is more general, even though it is assumed that they are both more general than 6.

(iii) *Non-intuitiveness of lazy reduction.* The unification procedure should yield a result that is consistent with the programmer's expectations. The behavior of reduction to type is not easy to anticipate in general, and in the context of unification it may simply be opaque. For example, what is the most general unifier of the unification problem

$$(x, y, x) = f(g(2))$$

in which f and g are both reducible functions and x and y are variables? If we take into account that the definition of f may or may not force the functional reduction of its argument $g(2)$, that the definition of f may or may not need to functionally reduce each element of the list it returns in order to determine that there are exactly three elements, and that the first and third elements may or may not need further reduction, it is likely to become a overly difficult problem for the programmer to anticipate what functional reductions will or will not be made as part of solving this very simple unification problem.

Given these objections, we instead choose to separate the notion of generality among unifiers, which will be examined immediately below, from that of laziness, which is discussed in a later section.

Most general unifiers in Juniper-B1

In the earlier definition of most general unifiers, a unique most general unifier was defined as the "top" of the generality ordering. Our approach will, instead, be to define the same kind of generality ordering, but extending each unifier in the earlier ordering to an *equivalence classes* of unifiers. The elements of a particular equivalence class differ at most by the fact that, for each variable, values bound to the variable are reducible to a common expression.

The implication of this approach is that unification problems do not, in Juniper-B1, have a single most general unifier, but a most general class of unifiers that, roughly speaking, are the same modulo functional reduction of the values in their bindings. The unification algorithm will return a single member of that most general class, and it will be of no concern, in terms of generality, which member it returns.

Note that this solution addresses the issues raised in the previous section by 1) excluding the issue of laziness altogether and 2) drawing a looser boundary around what we will accept as a most general unifier. The "looseness" is reflected in the fact that for a unification problem

$$x = f(1)$$

the most general unifier $\theta = \{ g(2)/x \}$ is just as suitable as $\theta' = \{ f(1)/x \}$, assuming that $f(1)$ and $g(2)$ reduce functionally to the same value, say the integer 6. While this may seem counter-intuitive at first, note that it is also consistent with our notion of satisfying the equations in a unification problem, e.g., the unifier $\theta = \{ g(2)/x \}$ would be said to satisfy the unification problem

$$x = f(1)$$

if there was a terminating computation showing that the value of $g(2)$ was the same as the value of $f(1)$.

We now proceed to define most general unifiers more formally. The generality relation on substitutions defined earlier, i.e., that the substitution θ is more general than a substitution θ' if there exists a substitution σ such that $\theta = \sigma.\theta'$, induces a partial order on the substitutions. Recall that all substitutions θ are total functions for which $\theta(x_i) = x_i$ for almost all i . Then

- θ Gen θ' if there exists a substitution σ such that $\theta = \sigma.\theta'$; or, in other words, if there exist one or more values t_i' for the bindings of variables x_i in θ' such that the t_i' 's are instances of the corresponding values t_i bound to x_i in θ .
- θ Gen θ'' if there exists θ''' such that θ Gen θ''' and θ''' Gen θ''

We next define Red to capture the "functionally reducible to" relation,

$t \text{ Red } t'$ if t and t' are grounded, and the functional reduction of t terminates with value t' .
 $f(t_1, \dots, t_n) \text{ Red } f'(t'_1, \dots, t'_n)$ if f and f' are constructors, $f = f'$, and either $t_i = t'_i$ or $t_i \text{ Red } t'_i$, $1 \leq i \leq n$.

The first assertion alone is not sufficient because of our reduction-to-type functional reduction policy - we wish also to relate terms on the left-hand side with "more fully than reduced-to-type" values on the right-hand side where possible. The second assertion, then, captures transitivity over reduction to type, which amounts to reduction of sub-terms.

What we need next, in order to define the proper equivalence classes, is a relation Red' which is the reflexive, symmetric, transitive closure of Red. The Red' relation is exactly the equality relation required for defining satisfaction of the equations in a unification problem, as discussed above. The definition is given by

$t \text{ Red}' t$ for all terms t .
 $t \text{ Red}' t'$ if $t \text{ Red } t'$ or $t' \text{ Red } t$.
 $t \text{ Red}' t'$ if there exists t'' such that $t \text{ Red}' t''$ and $t'' \text{ Red}' t'$.

The equivalence class of Red' containing t will be signified by $[t]_{\text{Red}'}$.

Note that $[t]_{\text{Red}'}$ does not necessarily contain the fully reduced form of t , since functional reduction of t or one of its subterms may not terminate.

We are now prepared to define the equivalence classes for Juniper-B1 unifiers:

$\theta \text{ Unif } \theta$ for all unifiers θ
 $\theta \text{ Unif } \theta'$ if, for each t_i bound to x_i in θ and t'_i bound to x_i in θ' , $t'_i \in [t_i]_{\text{Red}'}$.

Transitivity of Unif, follows from transitivity of Red'.

With this structure in place, it is a simple matter to define the Juniper-B1 generality relation, Gen_J:

$\theta \text{ Gen}_J \theta'$ if there exists θ_R and θ_R' such that $\theta_R \in [\theta]_{\text{Unif}_J}$ and $\theta_R' \in [\theta']_{\text{Unif}_J}$, and $\theta_R \text{ Gen } \theta_R'$.

Transitivity of Gen_J follows from transitivity of Unif_J .

A unifier θ is a *most general unifier* unless there exists a unifier θ' such that $\theta' \text{ Gen}_J \theta$, where θ' is not in $[\theta]_{\text{Unif}_J}$.

In our algorithm, as in Martelli and Montanari's, all reduction rules are unifier-preserving and unifiability will correspond exactly to termination with a solved system. Consequently the class of most general unifiers is also unique.

C.2.5. Juniper-B1 transformation rules

The Juniper-B1 unification algorithm will use the two transformation rules introduced earlier (T1 and T2), as well as a third that reflects the reducibility of function applications:

T3 (*Reduction to type*): For any equation of the form

$$t_1 = t_2$$

where t_1 is a grounded function application and t_2 is any non-variable term, replace the equation with

$$t_1' = t_2$$

where t_1' is t_1 reduced to type.

C.2.6. Juniper-B1 unification algorithm

The Juniper-B1 unification algorithm is an extension of Martelli and Montanari's. The algorithm consists of six reduction rules, four of which are taken directly from the earlier algorithm.

J1a (Constructor decomposition), analogue of R1: For any equation of the form

$$f(t_1, \dots, t_n) = f''(t''_1, \dots, t''_n)$$

where f' and f'' are n -ary constructors, if f' is not equal to f'' then halt with failure; otherwise apply the term decomposition transformation T1.

J1b (Function application reduction): For any equation of the form

$$f'(t_1, \dots, t_n) = t$$

where $f'(t_1, \dots, t_n)$ is a grounded n -ary function application and t is a non-variable term, replace the equation with

$$t'' = t$$

where t'' is the value of $f'(t_1, \dots, t_n)$ reduced to type.

J2 (Variable elimination), analogue of R2: For any equation of the form

$$z = t$$

where z is a variable and t is a term, if z occurs in t then halt with failure; otherwise if z occurs in another equation, apply the variable elimination transformation T2.

J3a (Left ordering for variables), analogue of R3: For any equation of the form

$$t = z$$

where z is a variable and t is not, replace the equation with

$$z = t$$

J3b (Left ordering for function applications): For any equation of the form

$$t = t'$$

where t is a construction or an ungrounded function application and t' is a grounded function application, replace the equation with

$$t' = t$$

J4 (*Variable identity elimination*), analogue of R4: For any equation of the form

$$s = s$$

where s is a variable, erase the equation.

C.2.7. Properties of Juniper-B1 unification

This section is devoted to characterizing the formal properties of the Juniper-B1 unification algorithm. In particular, we are concerned with termination, success and failure of unification, illegal programs and unification problems, and laziness of the algorithm presented.

Termination and Juniper-B1 unification

The Juniper-B1 unification algorithm can produce four different results:

1. It can halt with a unification problem in solved form;
2. It can halt with a unification problem not in solved form;
3. It can halt with failure;
4. It can fail to halt.

We will first show that case 1 corresponds to successful unification, case 2 corresponds to illegal programs and unification problems, and cases 3 and 4 correspond to unification failure, and are not in general distinguishable under arbitrary orderings of reduction rule applications.

Preservation of unifiers under transformation rules

The first step is to show, like Martelli and Montanari, that all transformation rules are unifier-preserving. Two theorems can be carried over directly from Martelli and Montanari's paper [1]:

Theorem 1 (MSM theorem 2.1): Let P be a unification problem and let

$$f(t_1, \dots, t_n) = f'(t'_1, \dots, t'_n)$$

be an equation of P , where f and f' are constructors. If f is not equal to f' then P has no unifier; otherwise, the new unification problem P' obtained by applying the constructor decomposition transformation $T1$ to the given equation has exactly the same unifiers as P .

Proof: If f is not equal to f' , then no substitution can make the two terms identical. If $f = f'$, any substitution which satisfies the equations of the decomposed term also satisfy the original equation, and conversely.

Theorem 2 (MSM theorem 2.2): Let P be a unification problem and P' be the unification problem obtained by applying the the variable elimination transformation $T2$ to an equation

$$z = t$$

in P , where z is a variable and t is any term not equal to z . If z appears in t , then P has no unifier; otherwise, P and P' are equivalent with respect to unifiers.

Proof: The equation

$$z = t$$

belongs to both P and P' , so any unifier θ must unify z and t . That is, $\theta(z)$ and $\theta(t)$ are identical. Now let

$$t_1 = t_2$$

be any other equation of S , and let

$$t'_1 = t'_2$$

be the corresponding equation in P' . Since t_1' and t_2' have been obtained by substituting t for every occurrence of z in t_1 and t_2 , respectively, we have $\theta(t_1) = \theta(t_1')$ and $\theta(t_2) = \theta(t_2')$. Thus, any unifier of P is also a unifier of P' and vice versa. Furthermore, if variable z occurs in t (but t is not z), then no substitution θ can make z and t identical, since $\theta(z)$ becomes a subterm of $\theta(t)$, and thus P has no unifier.

A similar theorem can be constructed for the T3 transformation rule:

Theorem 3: *Let P be a unification problem and P' be the unification problem resulting from a terminating application of the reduction-to-type transformation rule T3 to an equation*

$$t = t'$$

in P , where t is any grounded function application and t' is any non-variable term. Then P and P' have exactly the same unifiers.

Proof: Write the equation in P' resulting from T3 as

$$t'' = t'$$

t'' is t reduced to type. For any unifier θ of P , $\theta(t) = \theta(t')$ by hypothesis, and $\theta(t) = t$ since t is grounded. Then $t \in [t'']_{\text{red}}$ implies $\theta(t') = t = t'' = \theta(t'')$ since t'' is also grounded. Consequently, since θ unifies every other equation of P' , θ is a unifier of P' . Proof in the reverse direction is similar. Q.E.D.

It is trivial to show that unifiers are preserved under the transformations applied by the remaining reduction rules of the Juniper-B1 unification algorithm.

Termination and the Church-Rosser property

The next theorem has a strong correspondence to Martelli and Montanari's theorem 2.3(i):

Theorem 4: *The unification algorithm fails to halt if and only if an instance of functional reduction (transformation rule T3, applied via reduction rule J1b) fails to halt.*

Proof: An equivalent theorem, "The Juniper-B1 unification algorithm halts if and only if functional reduction halts", is easily proven in a fashion similar to Martelli and Montanari's theorem 2.3(i) (that their unification algorithm halts).

and then a theorem that is central to the characterisation of Juniper-B1 unification:

Theorem 5: *Given a unification problem P , if the Juniper-B1 unification algorithm terminates without failure for one ordering on the application of reduction rules, then it performs the same functional reductions irrespective of the order in which reduction rules are applied. Furthermore, if it terminates in solved form for one ordering, it terminates in solved form for any other ordering.*

Proof: *The proof of this theorem requires the development of a substantial amount of formal machinery. Consequently, the proof is deferred to Appendix A, where it is presented in full.*

Corollary 5.1: *Given a unification problem P , if the Juniper-B1 unification algorithm terminates in unsolved form for one ordering, it terminates in unsolved form for any other ordering.*

Proof: Follows directly from theorem 5.

Corollary 5.2 (Church-Rosser property): *Given a unification problem P , if the Juniper-B1 unification algorithm terminates in solved form under one ordering on the application of reduction rules, then it terminates in solved form under any other ordering, and yields a member of the equivalence class of unifiers which is most general under the Gen_γ generality relation.*

Proof: Termination in solved form follows from the premise and theorems 5 and 4. That the resulting unifier is in the class of most general unifiers follows immediately from the definitions of solved form and most general unifiers.

We need to further characterize the second class listed above, that in which the algorithm terminates in unsolved form:

Theorem 6: *If the unification algorithm halts in unsolved form then the initial unification problem is illegal.*

Proof: By inspection of the reduction rules and all possible pairings of terms (variable, constructor, grounded functional, ungrounded functional) in an equation, it is seen that there are only two cases in which an equation is both disallowed from

the solved form of a unification problem and uneliminable via the reduction rules: these are the cases in which one term is an ungrounded function application and the other is either a constructor or an ungrounded function application. Clearly, reducing equalities such as these is a necessary part of the matching operation performed by Juniper-B1 unification.

Since Juniper-B1 unification requires that there be an ordering of reduction rule applications under which each function application is grounded at the time its reduction is demanded, and since there is no such ordering if the algorithm halts in unsolved form under any ordering, by theorem 5, then the unsolved form is an illegal Juniper-B1 unification problem. Further, since theorems 1-3 assure that the unifiers of a problem are preserved under all transformation rules, the initial unification problem is illegal as well. Q.E.D.

Theorem 7: Termination with failure and non-termination of the unification algorithm are not in general distinguishable under variant orderings of reduction rule application.

Proof: Consider the unification problem

$$\begin{aligned}x &= 1 \\x &= 2 \\x &= f(1)\end{aligned}$$

where reduction to type of $f(1)$ is non-terminating. Clearly the problem is not satisfiable. The algorithm will terminate with failure if 1 is substituted for x and the 1=2 equality is tested, but will always fail to terminate if $f(1)$ is substituted for x .

Theorem 8: The unification algorithm halts in solved form if and only if the initial problem is legal and unifiable.

Proof: Forward direction: by corollary 5.2 and definitions of solved form and most general unifier. Reverse direction: the algorithm halts with failure only if the occurs check fails (J2) or an inequality is detected (J1a). If the algorithm fails to halt, there is no ordering under which it will halt without failure, by corollary 5.2. If it halts in unsolved form, the initial problem is illegal, by theorem 6. Hence it must halt in solved form.

In summary, what we have shown that the Juniper-B1 unification algorithm demonstrates three classes of behaviors corresponding to unifiable problems, illegal problems, and not unifiable problems. Furthermore, each of these classes is invariant with respect to choice of reduction rule orderings. These are summarized by

- a) The algorithm halts in solved form with a most general unifier if and only if the initial problem is legal in Juniper-B1 and unifiable;
- b) The algorithm halts in unsolved form if and only if the initial problem is illegal in Juniper-B1;
- c) Termination with failure and non-termination of the algorithm are not, in general, distinguishable. Both represent failure to unify in a legal Juniper-B1 unification problem.

Lasiness of Juniper-B1 unification

Our original requirement was that the Juniper-B1 unification algorithm perform functional reduction only when necessary to check for satisfaction of an equality (to do "matching"). The implication of theorem 5 is that all successful unifications perform the same reductions, e.g., the same as the most lazy ordering, one in which all functional reductions are delayed as long as possible. There is not a corresponding result for unification failure, since failure may be reflected either in termination of the algorithm with failure or failure to terminate.

C.2.8. Juniper-B1 unification and concurrent reduction semantics

The fact that failure of unification may be reflected in termination with failure or in failure to terminate may be a cause of some concern. In fact, from the viewpoint of the Juniper-B1 language as a whole, this is less problematic than might otherwise be supposed.

First, while the halting problem is known to be unsolvable, programmers generally have the ability to write terminating function applications, and non-terminating functional reduction most often will reflect errors in a program rather than the necessary but undesirable computational behavior of a correct program in a particular execution environment.

Second, Juniper-B1's concurrent reduction semantics accommodate non-termination better than most languages would. For example, if one is attempting to prove a goal using a particular logic-programming clause, and the head of the clause is not unifiable with the goal, the program will progress and produce solutions irrespective of whether unification with that clause terminates with failure or fails to terminate. It is still true, of course, that an attempt to explicitly collect all solutions will halt only if the non-unifiability produces termination with failure.

References

1. Martelli, A. and Montanari, U., An efficient unification algorithm, *ACM Trans. Prog. Lang. and Systems* 4, 2 (April, 1982), 258-282, ACM.

Appendix C-A: Proof of theorem 5

The proof of theorem 5 is not yet written up. It is based on a semantic proof procedure, and is consequently moderately messy.

Appendix D

Partial Evaluation of Juniper-B1 Programs

by

**Massoud Farhang
Bill Hopkins**

Table of Contents

Section	Title	Page
D.1	Approach	D-1
D.2	Example of Optimisations Obtained	D-1
D.3	Extended combinator reduction rules	D-2

Appendix D

Partial Evaluation of Juniper-B1 Programs

This informal note captures the significant aspects of the compile-time optimization of Juniper-B1 that was implemented at the end of the MASC Program.

D.1. Approach

We have defined and implemented a partial evaluator for the functional part of Juniper-B1, and implemented support for partial evaluation of the logical part of the language, including the implementation of unification by adding some primitive functions to the system. One key element of the partial evaluator is a function expansion strategy that recognises recursion and stops expansion after the first expansion, preventing an infinite loop of expansions.

The partial evaluator is a source-to-source program transformer. During partial evaluation of a logic expression, logic variables can be overwritten by compile-time unification implemented with gadgets. After partial evaluation, a functional expression equivalent to the original can be constructed, abstracting out variables that have not been overwritten and symbolic arguments from the definition that are still present.

The partial evaluator is an extension of a combinator-based implementation of Juniper-B1, described in Section 2.4 of this report. The partial evaluator uses an extended set of combinator reduction rules that are applied to the standard compiled form of the program. Undefined function arguments are replaced with special "unevaluable" constants; any attempt to reduce it to type will fail. If it occurs as a strict argument of a combinator, requiring that it be reduced to type prior reduction of to the combinator, the combinator expression will not be reduced. This effect ripples upward in the combinator graph to the extent that the subgraphs represent strict combinator arguments. By trying all possible reductions, however, many expressions are found not to be globally required, and irreducible expressions are often deleted by the reduction of non-strict combinators. The argument distribution combinators, for example (S, K, I, etc.) are non-strict, and the K combinator reduction often deletes irreducible arguments from expressions which then become reducible.

D.2. Example of Optimisations Obtained

The following example hints at the power of the reducer, demonstrating the ability to delete impossible cases from the optimised function application and to evaluate expressions within the function. It also demonstrates some limitations of the current implementation.

Given the function definition

DEF

$$f\ 1\ 2 = 2 * 3 + 4 \quad (1)$$

$$f\ 3\ n = n * (2 + 3) \quad (2)$$

$$f\ x\ y = x + y \quad (3)$$

we get the following compilation-time transformations:

$$f\ 5\ m \rightarrow 5 + m \quad (\text{by } 3)$$

$$f\ 1\ x \rightarrow f1\ x\ \text{WHERE}$$

$$f1\ 2 = 10 \quad (\text{by } 1)$$

$$f1\ y = 1 + y \quad (\text{by } 3)$$

$$f\ y\ 2 \rightarrow f2\ y\ \text{WHERE}$$

$$f2\ 1 = 10 \quad (\text{by } 1)$$

$$f2\ 3 = 3 * 5 \quad (\text{by } 2)$$

$$f2\ x = x + 2 \quad (\text{by } 3)$$

$$f\ y\ 7 \rightarrow f3\ y\ \text{WHERE}$$

$$f3\ 3 = 3 * 7 \quad (\text{by } 2)$$

$$f3\ x = x + y \quad (\text{by } 3)$$

The results in this area are rather preliminary, and have not been exhaustively tested. Several of the examples contain unreduced arithmetic expression, indicating that the control mechanisms have not been optimally tuned.

D.3. Extended combinator reduction rules

In this section we document the extensions for partial evaluation to the basic fixed combinator reduction rules. We have also included some simple optimization rules for recognizing and simplifying identity relations in arithmetic and Boolean logic expressions.

In the following, ?exp stands for a partially evaluated exp.

$$S\ f\ g\ x \rightarrow (f\ x)(g\ x)$$

$$C\ f\ g\ x \rightarrow (f\ x)\ g$$

$$B\ f\ g\ x \rightarrow f\ (g\ x)$$

$$K\ x\ y \rightarrow x$$

$$\text{MATCH } a\ \text{FAIL } b \rightarrow \text{FAIL}$$

MATCH a f a $\rightarrow$ f
 MATCH a f b $\rightarrow$ FAIL, if a $\neq$ b
 MATCH a f b $\rightarrow$ MATCH a f b, if ?a and/or ?b

 Y h $\rightarrow$ Y h

 I x $\rightarrow$ x

 S1 k f g x $\rightarrow$ k (f x) (g x)

 C1 k f g x $\rightarrow$ k (f x) g

 B1 k f g x $\rightarrow$ k (f (g x))

 HD (a:b) $\rightarrow$ a
 HD other $\rightarrow$ HD other

 TL (a:b) $\rightarrow$ b
 TL other $\rightarrow$ TL other

 S_p f g x $\rightarrow$ (f x) : (g x)

 B_p f g x $\rightarrow$ f : (g x)

 C_p f g x $\rightarrow$ (f x) : g

 U f x $\rightarrow$ f (HD x) (TL x)

 U_ f (a:b) $\rightarrow$ f a b
 U_ f other $\rightarrow$ FAIL, if other of type CONSTRUCTOR or NUM
 U_ f ?l $\rightarrow$ U_ f ?l

 U_a f y $\rightarrow$ FAIL, if y of type CONSTRUCTOR, CONS or NUM
 U_a f l $\rightarrow$ U_a f l

 TRY f g x $\rightarrow$ TRY (f x) (g x)
 TRY (FAIL ...) g $\rightarrow$ g
 TRY f (FAIL ...) $\rightarrow$ f
 TRY f g $\rightarrow$ f, if f is a NUM or a CONSTRUCTOR
 TRY f g $\rightarrow$ TRY f g

 STARTREAD filename $\rightarrow$ READ filedescriptor

READ fledes $\rightarrow$ nextchar : READ fledes
READ file, EOF $\rightarrow$ NIL

COND TRUE $x\ y \rightarrow x$
COND FALSE $x\ y \rightarrow y$
COND $b\ x\ y \rightarrow$ COND $b\ x\ y$

EQ $x\ x \rightarrow$ TRUE
EQ $x\ y \rightarrow$ FALSE, if $x \neq y$
EQ $x\ y \rightarrow$ EQ $x\ y$, if ? x and/or ? y

APPEND ($a:x$) $y \rightarrow a : (APPEND\ x\ y)$
APPEND NIL $y \rightarrow y$
APPEND x NIL $\rightarrow x$
APPEND ! $y \rightarrow$ APPEND ! y

AND FALSE $y \rightarrow$ FALSE
AND x FALSE $\rightarrow$ FALSE
AND $x\ y \rightarrow y$, if x is reduced to type and $x \neq$ FALSE
AND $x\ y \rightarrow x$, if y is reduced to type and $y \neq$ FALSE
AND $x\ y \rightarrow$ AND $x\ y$

OR TRUE $y \rightarrow$ TRUE
OR x TRUE $\rightarrow$ TRUE
OR $x\ y \rightarrow y$, if x is reduced to type and $x \neq$ TRUE
OR $x\ y \rightarrow x$, if y is reduced to type and $y \neq$ TRUE
OR $x\ y \rightarrow$ OR $x\ y$

NOT TRUE $\rightarrow$ FALSE
NOT FALSE $\rightarrow$ TRUE
NOT $x \rightarrow$ NOT x

NEG $x \rightarrow -x$, if x is a NUM
NEG $x \rightarrow$ NEG x

GR $x\ y \rightarrow x > y$, if x and y are NUMs
GR $x\ y \rightarrow$ GR $x\ y$

GRE $x\ y \rightarrow x \geq y$, if x and y are NUMs
GRE $x\ y \rightarrow$ GRE $x\ y$

MUCHGR $x\ y \rightarrow x \gg y$, if x and y are NUMs
MUCHGR $x\ y \rightarrow$ MUCHGR $x\ y$

MINUS $x\ y \rightarrow x - y$, if x and y are NUMs

MINUS $0\ y \rightarrow -y$

MINUS $x\ 0 \rightarrow x$

MINUS $x\ y \rightarrow \text{MINUS } x\ y$

PLUS $x\ y \rightarrow x + y$, if x and y are NUMs

PLUS $0\ y \rightarrow y$

PLUS $x\ 0 \rightarrow x$

PLUS $x\ y \rightarrow \text{PLUS } x\ y$

TIMES $x\ y \rightarrow x * y$, if x and y are NUMs

TIMES $1\ y \rightarrow y$

TIMES $x\ 1 \rightarrow x$

TIMES $0\ y \rightarrow 0$

TIMES $x\ 0 \rightarrow 0$

TIMES $x\ y \rightarrow \text{TIMES } x\ y$

INTDIV $x\ y \rightarrow \text{integer-part}(x / y)$, if x and y are NUMs

INTDIV $x\ y \rightarrow \text{INTDIV } x\ y$

FDIV $x\ y \rightarrow x / y$, if x and y are NUMs

FDIV $x\ 1 \rightarrow x$

FDIV $x\ y \rightarrow \text{FDIV } x\ y$

MOD $x\ y \rightarrow \text{mod}(x, y)$, if x and y are NUMs

MOD $x\ y \rightarrow \text{MOD } x\ y$

POWER $x\ y \rightarrow x ** y$, if x and y are NUMs

POWER $1\ y \rightarrow 1$

POWER $x\ y \rightarrow \text{POWER } x\ y$

ALLOCATE $n\ r\_proc \rightarrow r\_proc\ \text{alloc1} \dots \text{allocn}$

COPY_UNBOUND $(r\_proc\ \text{cont}\ \text{arg1} \dots \text{argn}) \rightarrow (r\_proc\ \text{cont}'\ \text{arg1}' \dots \text{argn}')$

APPLY $f\ [] \rightarrow f$

APPLY $f\ [x:y] \rightarrow \text{APPLY } (f\ x)\ y$

GADGETIZE $\text{logic\_unb\_var\_x} \rightarrow \text{VAR\_GADGET } \text{logic\_unb\_var\_x}$

GADGETIZE $\text{number\_i} \rightarrow \text{NULLARY\_GADGET } \text{number\_i}$

GADGETIZE $f\ (<\text{args}>) \rightarrow \text{GADGETIZE } \text{reduced}(f\ (<\text{args}>))$

GADGETIZE $\text{struct } ([A|B]) \rightarrow \text{UNARY\_GADGET } \text{struct } (\text{GADGETIZE } [A|B])$

GADGETIZE $[A|B] \rightarrow$

BINARY_GADGET $\text{cons } (\text{GADGETIZE } A)\ (\text{GADGETIZE } [B])$

```

UNGADGETIZE logic_unb_var_x → logic_unb_var_x
UNGADGETIZE (VAR_GADGET X) → X
UNGADGETIZE (NULLARY_GADGET number_i) → number_i
UNGADGETIZE (UNARY_GADGET struct <args>) →
    (APPLY struct (UNGADGETIZE <args>))
UNGADGETIZE (BINARY_GADGET cons <hd> <tl>) →
    [(UNGADGETIZE <hd>)](UNGADGETIZE <tl>)]

VAR_GADGET logic_unb_var_x cont gadget.imp →
    [gadget.exp/logic_unb_var_x] cont
VAR_GADGET gadget.exp cont gadget.imp → gadget.exp cont gadget.imp
VAR_GADGET logic_var_x cont (VAR_GADGET logic_var_x) → cont
VAR_GADGET logic_unb_var_x cont (VAR_GADGET logic_var_y) →
    [(VAR_GADGET logic_var_y)/logic_unb_var_x] cont

NULLARY_GADGET value1 cont $0:value1 → cont
NULLARY_GADGET value1 cont $0:value2 → NIL
NULLARY_GADGET a b $1:struct → NIL
NULLARY_GADGET a b $2:constr → NIL
NULLARY_GADGET value cont gadget → (gadget cont $0:value)

UNARY_GADGET struct1 args1 [args2.cont] $1:struct1 → (args1 cont args2)
UNARY_GADGET struct1 args1 [args2.cont] $1:struct2 → NIL
UNARY_GADGET a b c $0:value → NIL
UNARY_GADGET a b c $2:constr → NIL
UNARY_GADGET struct args cont gadget → (gadget [args.cont] $1:struct)

BINARY_GADGET constr1 hd1 tl1 [hd2.tl2.cont] $2:constr1 →
    (hd1 (tl1 cont tl2) hd2)
BINARY_GADGET constr1 hd1 tl1 [hd2.tl2.cont] $2:constr2 → NIL
BINARY_GADGET a b c d $0:value → NIL
BINARY_GADGET a b c d $1:struct → NIL
BINARY_GADGET constr hd tl cont gadget → (gadget [hd.tl.cont] $2:constr)

```

DISTRIBUTION LIST

addresses	number of copies
Robert L. Kaminski RADC/COTC	5
RADC/BOVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STAGE 5 ALEXANDRIA VA 22304-6145	5
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
Director DMAAC (Attn: RE) 3200 S. Second St. St Louis MO 63118-3399	1
AFCSA/SAHI Attn: Miss Griffin 10363 Pentagon Wash DC 20330-5425	1

HQ USAF/SCTT Pentagon Wash DC 20330-5190	1
SAF/AQSC Pentagon 4D-267 Wash. DC 20330-1000	1
DIRECTOR DMAHTC ATTN: SDSIM Wash DC 20315-0030	1
Director, Info Systems OASD (C3I) Rm 3E187 Pentagon Wash DC 20301-3040	1
Fleet Analysis Center Attn: GIDEP Operations Center Code 3061 (E. Richards) Corona CA 91720	1
HQ AFSC/XTKX ANDREWS AFB DC 20334-5000	1
HQ AFSC/XRT Andrews AFB MD 20334-5000	1
HQ AFSC/XRK ANDREWS AFB MD 20334-500	1
HQ SAC/NRI OF FUTT AFB NE 68113-5001	1

HQ SAC/SCPT
OFUTT AFB NE 68113-5001

1

HQ ESB/DOQR
Attn: Fred Ladwig
San Antonio TX 78243-5000

1

DTESA/RQEE
ATTN: LARRY G. MC MANUS
2501 YALE STREET SE
Airport Plaza, Suite 102
ALBUQUERQUE NM 87106

1

HQ TAC/DRIY
Attn: Mr. Westerman
Langley AFB VA 23665-5001

1

HQ TAC/DOA
LANGLEY AFB VA 23665-5001

1

HQ TAC/DRCA
LANGLEY AFB VA 23665-5001

1

ASD/ENEMS
Wright-Patterson AFB OH 45433-6503

2

ASD/AFALC/AXAE
Attn: W. H. Dungey
Wright-Patterson AFB OH 45433-6533

1

ASD/ENACA
Wright-Patterson AFB OH 45433

1

WRDC/AAAI 1
WRIGHT-PATTERSON AFB OH 45433-6533

AFIT/LDEE 1
BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

WRDC/MLPO 1
WRIGHT-PATTERSON AFB OH 45433-6533

WRDC/MLTE 1
WRIGHT-PATTERSON AFB OH 45433

WRDC/FIES/SURVIAC 1
WRIGHT-PATTERSON AFB OH 45433

AAMRL/HE 1
WRIGHT-PATTERSON AFB OH 45433-6573

Air Force Human Resources Laboratory 1
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433

2750 ABW/SSLT 1
Bldg 262
Post 11S
Wright-Patterson AFB OH 45433

AFHRL/OTS
WILLIAMS AFB AZ 85240-6457

1

AUL/LSE
MAXWELL AFB AL 36112-5564

1

HQ AFSPACECOM/XPYS
ATTN: DR. WILLIAM R. MATOUSH
PETERSON AFB CO 80914-5001

1

3280TTG/BISS
Attn: TSgt Kirk
Lackland AFB TX 78236

1

Defense Communications Engineering Ctr
Technical Library
1860 Wiehle Avenue
Reston VA 22090-5500

1

COMMAND CONTROL AND COMMUNICATIONS DIV
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION COMMAND
ATTN: CODE DIOA
QUANTICO VA 22134-5080

2

AFLMC/LGY
ATTN: CH, SYS ENGR DIV
GUNTER AFS AL 36114

1

U.S. Army Strategic Defense Command
Attn: DASD-H-MPL
P.O. Box 1500
Huntsville AL 35807-3801

1

COMMANDING OFFICER
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46215-2189

1

COMMANDING OFFICER 1
NAVAL TRAINING SYSTEMS CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100

COMMANDER 1
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 96428
SAN DIEGO CA 92152-5000

COMMANDER (CODE 3433) 1
ATTN: TECHNICAL LIBRARY
NAVAL WEAPONS CENTER
CHINA LAKE, CALIFORNIA 93555-6001

SUPERINTENDENT (CODE 1424) 1
NAVLA POST GRADUATE SCHOOL
MONTEREY CA 93943-5000

COMMANDING OFFICER 2
NAVAL RESEARCH LABORATORY
ATTN: CODE 2627
WASHINGTON DC 20375-5000

SPACE & NAVAL WARFARE SYSTEMS COMMAND 1
PMW 153-3DP
ATTN: R. SAVARESE
WASHINGTON DC 20363-5100

CDR, U.S. ARMY MISSILE COMMAND 2
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN: APSMI-RD-CS-R (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5241

Advisory Group on Electron Devices 2
Hammond John/Technical Info Coordinator
201 Varick Street, Suite 1140
New York NY 10014

UNIVERSITY OF CALIFORNIA/LOS ALAMOS 1
NATIONAL LABORATORY
ATTN: DAN BACA/REPORT LIBRARIAN
P.O. BOX 1663, MS-P364
LOS ALAMOS NM 87545

RAND CORPORATION THE/LIBRARY 1
HELPER DORIS S/HEAD TECH SVCS
P.O. BOX 2138
SANTA MONICA CA 90406-2138

AEDC LIBRARY (TECH REPORTS FILE) 1
MS-100
ARNOLD AFS TN 37389-9998

USAG 1
Attn: ASH-PCA-CRT
Ft Huachuca AZ 85613-6000

1839 EIG/EIET (KENNETH W. IRBY) 1
KEESLER AFB MS 39534-6348

JTFPMO 1
Attn: Director/Advanced Technology
1500 Planning Research Drive
McLean VA 22102-5099

AWS TECHNICAL LIBRARY 1
FL4414
SCOTT AFB IL 62225-5458

HQ ESC/CWPP 1
San Antonio TX 78243-5000

AFEWC/ESRI 3
SAN ANTONIO TX 78243-5000

485 EIG/EIER (DMO) 2
GRIFFISS AFB NY 13441-6348

ESD/AVS
ATTN: ADV SYS DEV
HANSCom AFB MA 01731-5000

1

ESD/ICP
HANSCom AFB MA 01731-5000

1

ESD/AVSE
BLDG 1704
HANSCom AFB MA 01731-5000

2

HQ ESD SYS-2
HANSCom AFB MA 01731-5000

1

ESD/TCD-2
ATTN: CAPTAIN J. MEYER
HANSCom AFB MA 01731-5000

1

The Software Engineering Institute
Attn: Major Dan Burton, USAF
Joint Program Office
Carnegie Mellon University
Pittsburgh PA 15213-3890

1

DIRECTOR
NSA/CSS
ATTN: T513/TDL (DAVID MARJARUM)
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: W166
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R-8316 (MR. ALLEY)
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R24
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R21
9800 SAVAGE ROAD
FORT GEORGE G MEASDE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: DEFSMAC (JAMES E. HILLMAN)
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R5
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R8
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: S031
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: S21
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: V33 (S. Friedrich)
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: W07
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: W3
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R523
FORT GEORGE G MEADE MD 20755-6000

2

DoD COMPUTER SECURITY CENTER
ATTN: C4/TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000

1

attn: William C. Hopkins
UNISYS Paoli Research Center
P. O. Box 517
Paoli, PA 19301

5

Defense Advanced Research Projects Agency (ISTO)
1400 Wilson Blvd
Arlington VA 22209-2308

1