AFOSR-TR- 78-1091

LEVEL II

AD A056071

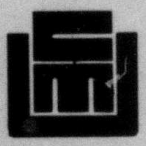
Mechanical Theorem Proving and Artificial Intelligence Languages

Mark E. Stickel

December 6, 1977

DDC
JUL 6 1978
F

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

AD No.
DDC FILE COPY

78 06 27 090

Approved for public release;
distribution unlimited.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-78-1091	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER: Doctoral thesis.
4. TITLE (and Subtitle): MECHANICAL THEOREM PROVING AND ARTIFICIAL INTELLIGENCE LANGUAGES.	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Mark E. Stickel	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101E, A02466/7, 2466
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: Dec 1977
	13. NUMBER OF PAGES: 352
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research (NM), Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This dissertation is principally concerned with incompleteness issues in the design of artificial intelligence languages. Major sources of incompleteness are the pattern matching the inference facilities of the languages. Incompleteness in the area of pattern matching can be repaired by developing unification algorithms for the specialized data types of the languages. A complete, but potentially infinite unification process is described for arbitrary data types in general and is applied to the QA4/QLISP vector, bag, and class data types. Finite, complete unification algorithms are also described for the bag and class data types. The bag unifica-

continued

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081

UNCLASSIFIED

20. abstract (continued)

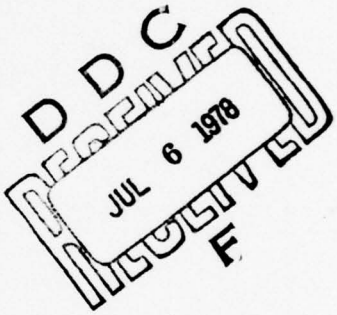
tion algorithm is extended to the case of unification of first order predicate calculus terms with functions which are both associative and commutative. Incompleteness in the area of the inference system can be repaired by use of some form of the inference procedure which is a complete extension derived from model elimination of the problem reduction method. This can readily be accomplished in present or new artificial intelligence languages by attempting to derive all goals in the context of the asserted negations of all high goals. The problem of compatibility of the procedure and use of models is addressed though not finally resolved. Design of the experimental results for the theorem proving program PSTP are presented. The inference system used is a variant of the π procedure and a version of PSTP was used to verify the effectiveness of the special unification algorithms developed.

LEVEL II

Mechanical Theorem Proving and Artificial Intelligence Languages

Mark E. Stickel

December 6, 1977



Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This document has been approved for public release and sale; its distribution is unlimited.

This research was supported in part by the National Science Foundation (GJ-28457) and by the Advanced Research Projects Agency of the Secretary of the Office of Defense (F44620-73-C-0074).

Abstract

This dissertation is principally concerned with incompleteness issues in the design of artificial intelligence languages. Major sources of incompleteness are the pattern matching and inference facilities of the languages. Incompleteness in the area of pattern matching can be repaired by developing unification algorithms for the specialized data types of the languages. A complete, but potentially infinite unification process is described for arbitrary data types in general and is applied to the QA4/QLISP vector, bag, and class data types. Finite, complete unification algorithms are also described for the bag and class data types. The bag unification algorithm is extended to the case of unification of first order predicate calculus terms with functions which are both associative and commutative. Incompleteness in the area of the inference system can be repaired by use of some form of the π inference procedure which is a complete extension derived from model elimination of the problem reduction method. This can readily be accomplished in present or new artificial intelligence languages by attempting to derive all goals in the context of the asserted negations of all higher goals. The problem of compatibility of the π procedure and use of models is addressed though not finally resolved. Design of and experimental results for the theorem proving program PSTP are presented. The inference system used is a variant of the π procedure and a version of PSTP was used to verify the effectiveness of the special unification algorithms developed.

78 06 27 090

Acknowledgments

I would like to thank the members of my committee, Professors Buchanan (Chairman), Andrews, Loveland, Newell, and Simon for their assistance to me and confidence in me during the preparation of this dissertation. Their criticisms were constructive, appropriate, and prompt, and signicantly improved the dissertation.

I am especially grateful to Professor Loveland for serving as my advisor during the first years I was at CMU and for teaching me so much of what I know about mechanical theorem proving; to Professor Buchanan for serving as my advisor during the later years, assisting me through the process of dissertation proposal, writing, and defense; and to Professor Andrews for his expertise and advice on mechanical theorem proving.

Finally, I would like to commend the Computer Science Department as a whole for providing such an excellent environment for learning and research.

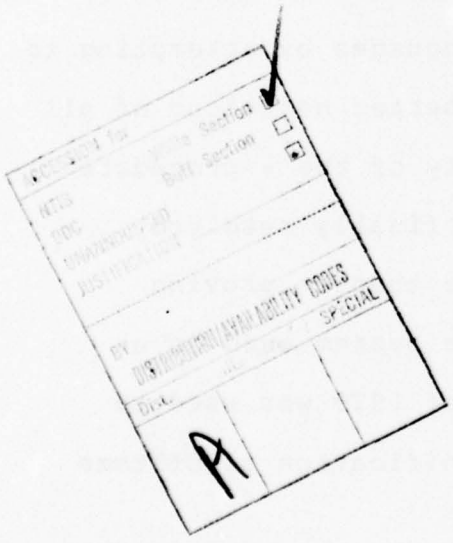


Table of Contents

1. Introduction	1
1.1. Artificial intelligence languages	1
1.2. Completeness	4
1.3. Control of theorem proving programs	10
1.4. Inconsistency and uncertainty	12
2. An Inference System for Artificial Intelligence Languages	20
2.1. Introduction	20
2.2. Terminology	24
2.3. Problem reduction method	27
2.4. Herbrand's theorem	31
2.5. Resolution theorem proving procedure	41
2.5.1. Binary resolution	41
2.5.2. Binary factorization	42
2.5.3. Raw resolution	42
2.5.4. Tautology elimination	43
2.5.5. Subsumption	43
2.5.6. Set of support strategy	44
2.5.7. Input resolution	45
2.5.8. Unit resolution	46
2.5.9. Linear resolution	47
2.5.10 Paramodulation	48
2.6. Model elimination theorem proving procedure	48
2.6.1. Admissability	50
2.6.2. Extension	50
2.6.3. Reduction	51
2.6.4. Strong model elimination	52
2.6.5. Acceptability	52
2.6.6. Extension	53
2.6.7. Reduction	53
2.6.8. Factorization	53
2.6.9. Model elimination with paramodulation	54
2.6.10. Permissability	54
2.6.11. Extension	55
2.6.12. Reduction	55
2.6.13. Factorization	55
2.6.14. P-extension	55
2.6.15. P-reduction	56
2.7. MESON theorem proving procedure	58
2.7.1. MESON procedure	60
2.7.2. MESON procedure for equality	61
2.8. Linear MESON theorem proving procedure	62
2.8.1. Matching	69
2.8.2. Extension	69
2.8.3. Factorization	70
2.8.4. Reduction	70
2.8.5. P-extension	70
2.8.6. P-reduction	72
2.8.7. Postprocessing	74
2.8.8. Lemma formation	78
2.8.9. Subsumption	80

2.9. The Programmable Strategy Theorem Prover 83
2.9.1. Chain properties 84
2.9.2. Chain filters 87
2.9.3. Search strategy specification 88
2.9.4. Chain order functions 89
2.9.5. Search commands 90
2.9.6. Format functions 93
2.9.7. Command summary 93
2.9.8. ANCESTRY command 94
2.9.9. CHAINLIST command 94
2.9.10. COPY command 95
2.9.11. COUNT command 95
2.9.12. DELETE command 95
2.9.13. DISPLAY command 95
2.9.14. EXPAND command 95
2.9.15. FOR command 96
2.9.16. PARAMETERS command 96
2.9.17. POSTPROCESSING command 97
2.9.18. PROBLEM command 97
2.9.19. SEARCH command 98
2.9.20. SEARCH2 command 99
2.9.21. TRANSFER command 100
2.10. PSTP performance study 100
2.10.1. The problems 100
2.10.2. Inference operations used 101
2.10.3. Search strategies used 101
2.10.4. Statistics 104
2.10.5. Results 108
2.11. π theorem proving procedure 116
2.12. Multiple world states 124
2.13. Equality 127
2.13.1. Marking and other restrictions 131
2.13.2. Demodulation 143
2.14. Use of models 148
2.14.1. Geometry-Theorem Proving Machine 149
2.14.2. Abstraction of Geometry-Theorem Proving Machine 153
2.14.3. Types of models 158
2.14.4. Incompatibility of π procedure with use of models 161
2.14.5. A model refinement for the π procedure 164

3. Unification Algorithms for Artificial Intelligence Languages 171
3.1. Introduction 171
3.2. Terminology 175
3.3. Semi-unification and variable splitting 183
3.4. Semi-unification 186
3.5. Vector semi-unification 189
3.6. Bag semi-unification 191
3.7. Class semi-unification 194
3.8. Examples of semi-unification 197
3.9. Termination and soundness of semi-unification 203
3.10. Variable splitting 213
3.11. Semi-unification and elaboration 216
3.12. Completeness of semi-unification and variable

splitting 222
3.13. Vector unification 238
3.14. Bag unification 244
3.15. AC unification 262
3.16. Class unification 275
3.17. Constraints 282
3.18. Algebraic unification 285
3.19. Complete sets of reductions 287
3.20. Implementation 292
3.21. Experimentation 295
3.22. Background and summary 303

4. Summary and Conclusion 308
4.1. An inference system for artificial intelligence languages 309
4.2. Programmable Strategy Theorem Prover 311
4.3. Unification algorithms for artificial intelligence languages 313

A. PSTP Performance Study Results 317
A.1. Examples 317
A.2. Statistics 320
A.3. Proofs 323

B. Associativity and Commutativity Test Results 327
B.1. Examples 327
B.2. Statistics 329
B.3. Proofs 331

References 341

1. Introduction

This thesis is broadly concerned with the areas of mechanical theorem proving and artificial intelligence languages.

First, we will present the design of and some experimental results for a particular theorem proving program: the Programmable Strategy Theorem Prover (PSTP). This program uses the newly developed linear MESON procedure as its inference system, and provides great flexibility in the user specification of search strategy for either interactive or fully automatic use.

The remainder of this thesis will be devoted to a discussion of completeness issues in the design of artificial intelligence languages. We will discuss several separate areas of incompleteness in these languages and remedies of the incompleteness, drawing upon the field of mechanical theorem proving for our solutions.

These two parts of this thesis are linked in that the design of and experience with PSTP motivated some of the ideas for remedying incompleteness in artificial intelligence languages, and ideas developed in consideration of artificial intelligence languages have been tested and incorporated into PSTP.

1.1. Artificial intelligence languages

We mean by the term artificial intelligence languages languages such as PLANNER [21], CONNIVER [33], and QA4/QLISP [43,37,53,44]. These are high level, problem solving languages

designed to be used as programming languages for artificial intelligence research. A survey of these and other languages for artificial intelligence research is provided by Bobrow and Raphael [7].

These languages can be characterized by and differentiated from other languages by possessing certain common characteristics.

First of these is some facility for pattern matching. Pattern matching facilities are provided for associative retrieval of data, for data item decomposition, and for data item composition. Patterns and data items are constructed of such things as constants (PLANNER, CONNIVER, QA4/QLISP), simple variables (PLANNER, CONNIVER, QA4/QLISP), fragment variables (PLANNER, QA4/QLISP, partially in CONNIVER), pattern matching actors or constraints (PLANNER, QA4/QLISP), lists (PLANNER, CONNIVER, QA4/QLISP), and bags and classes (QA4/QLISP).

The second major feature of artificial intelligence languages is a capability for automated deduction. There is a great deal of similarity in the deduction mechanisms of the various artificial intelligence languages. PLANNNER, CONNIVER, QA4/QLISP all use backward chaining via the GOAL statement and CONSEQUENT theorems, the FETCH statement and IF-NEEDED methods, and the GOAL statement and APPLY TEAM functions respectively. These essentially identical (except for terminology) mechanisms we consider to be direct use of the problem reduction method explicated by Nilsson [35]--reducing (in multiple ways to deal

with alternative possible solutions) a goal to a set of subgoals, each of which must be solved in order to solve the parent goal.

They also provide facilities for updating the world state maintained by the system when data items are added to or deleted from the data base. Again, these mechanisms are essentially identical (except for terminology) in PLANNER, CONNIVER, and QA4/QLISP. Each language provides the capability for performing arbitrary actions (but usually additions to or deletions from the data base, representing the making of subsidiary deductions from an altered state) when there are additions to or deletions from the data base. These actions are performed by the ANTECEDENT or ERASING theorems of PLANNER, by the IF-ADDED or IF-REMOVED methods of CONNIVER, and by APPLY TEAM functions of QLISP (demons in QA4).

The third principal feature characterizing artificial intelligence languages is special control structures, and the data structure required to support them.

The first of this generation of artificial intelligence languages, PLANNER, "introduced" backtracking as a mechanism for handling alternatives in the search for a solution in the problem reduction space. A limited parallelism was also allowed.

Reacting against the inefficiency of backtracking, CONNIVER provided a number of search control primitives, and made these accessible to the user.

QA4/QLISP also implemented backtracking, and provided coroutines to obtain some parallelism and non-backtracking alternation of search domain.

1.2. Completeness

What we mean by completeness of artificial intelligence languages requires some explanation.

In a very general sense, all these languages are complete, as are most other computer languages. There exist several definitions of computable functions: recursive function theory, Turing machines, Markov normal systems. They all turn out to define the same set of functions, and are believed (Turing's thesis, Church's thesis) to define the class of all computable functions. All the artificial intelligence languages are theoretically capable of computing these functions (they can all, for example, be used to program a Turing machine simulator).

However, this is a very weak definition of language completeness which almost entirely fails to distinguish between any computer languages: the weakest assembly language and PLANNER, and every language in between, are all complete in this sense.

The definition of completeness as applied to mechanical theorem proving procedures is that a procedure is complete if and only if (given sufficient resources (time and space)) it is capable of finding a proof for any properly posed theorem. Since a major part of what characterizes artificial intelligence languages is a facility for deduction, this definition is applicable to evaluating the completeness of artificial intelligence languages.

The connection between completeness of theorem proving procedures and artificial intelligence language completeness is most direct in the area of this deduction mechanism.

The principal difficulty of completeness with this deduction mechanism is not with the state updating operations, but with the problem reduction method itself. The problem reduction method is an adequate and complete deduction mechanism provided the logical negation of a goal has no meaning. However, as soon as negations of goals are permitted, the problem reduction method is incapable of performing some valid deductions.

We will discuss an extension of the problem reduction method which is logically complete for the first order predicate calculus. This method is essentially an abstraction of the linear MESON procedure used in PSTP.

We can regard a goal as representing a theorem to be proved; we can regard PLANNER CONSEQUENT theorems to be axioms; the pattern of the consequent theorem is the consequent of the axiom, the body is the antecedent. Our question concerning completeness can be posed as: if we encode axioms as CONSEQUENT theorems, and a theorem to be proved in this way, will PLANNER always discover a proof? The answer is no, for several reasons.

One reason is that the PLANNER deduction mechanism must, in general, have access to contrapositives of axioms as well as the axioms themselves. Thus, if we encode A → B as <CONSEQUENT [] [B] <GOAL [A]>>, we must also present <CONSEQUENT [] [NOT [A]] <GOAL [NOT [B]]>>. This we consider to be a responsibility of the user, and not a completeness deficiency of the language.

The second, more crucial reason is the logical incompleteness (relative to the first order predicate calculus, or the propositional calculus) of this deduction mechanism, which is merely an implementation of the problem reduction method.

The point is that the user can reasonably expect, or at least hope, that the deduction mechanism that PLANNER provides is complete--that it will solve the goal if it is logically possible to solve it.

The incompleteness of the deduction mechanism can be "remedied" in various ways. One way is to just ignore the problem, making it a user responsibility to program whatever additional inference mechanisms he needs for completeness. This is essentially the position of present artificial intelligence languages. Another solution is to provide within the language some backup, complete deductive facility. This is actually done in PLANNER, which has a resolution theorem proving subsection. A third solution is to alter the deduction mechanism so that its use now meets the user expectation of completeness directly, rather than using a separate facility to obtain completeness. This last approach is the one being explored in this thesis.

In general, our approach is to obtain completeness in a form compatible with present concepts and structures of artificial intelligence languages, and without significantly adversely affecting their applicability to and performance on present applications.

The second area of incompleteness is in pattern matching.

Since pattern matching in the deduction mechanism plays the role of unification in resolution, it is apparent that for completeness, pattern matching must be as general as unification, i.e., it must be able to represent every possible match.

Even disregarding the functional analogy to unification, it is a reasonable expectation that every possible match be returned by the pattern matching algorithms.

Pattern matching as customarily used uses an expression called the pattern expression (pattern) and another expression called the target expression (target). In its customary usage, pattern matching is not symmetric; the permitted structure differs for patterns and targets. In general, variables and pattern matching actors are restricted to appearing in patterns; they may not appear in targets. Thus, the task of pattern matching consists of finding assignments for variables and processing pattern matching actors such that the resulting instantiated pattern is identical (or equivalent)) to the target.

We will show that for some purposes, it is important that pattern matching be symmetric. Specificially, we will not want to restrict variables and actors to appearing in only one of the expressions, but will want to allow both expressions to be equally general.

So long as pattern matching severely limited the form of the target, pattern matching is a comparatively simple task, and completeness is easily maintained. However, as the specification of targets becomes more general, the pattern matching task

becomes progressively more difficult, ultimately reaching the stage where it is, in general, an undecidable problem to determine if two expressions have any common instances.

We will consider in detail the pattern matching of QA4/QLISP. CONNIVER pattern matching is either already complete, or could readily be made so. PLANNER patterns are general enough to create the problem of potential undecidability. The principal pattern matching facilities of QA4/QLISP provide an intermediate level of theoretical difficulty. Furthermore, they (vectors, bags, and classes) are important mathematical objects, and the study of pattern matching for them has significance beyond their immediate application to QA4/QLISP.

A third area of incompleteness in the control structure of programs written in these artificial intelligence languages. A reliance upon depth first search (with backtracking on failure) is necessarily incomplete whether the depth is unbounded (permitting infinite digressions not leading to a solution) or bounded (possibly eliminating all solutions due to extreme depth). It is essential for completeness that there exist the capability for maintaining simultaneously multiple potential solutions of arbitrary size. The searching procedures of PSTP (essentially general heuristic search) provide one solution to this problem.

For efficiency reasons, as well as incompleteness reasons, there has been a strong move against use of backtracking in later design and use of artificial intelligence languages. Though

practical matters remain (like how can one easily specify a search procedure), the theoretical issues are well understood and widely recognized, so we will have little to say about this area.

One contribution artificial intelligence languages have made is in this area of richer control structure. This thesis is concerned with the intersection of the areas of mechanical theorem proving and artificial intelligence languages. Mechanical theorem proving provides many insights into how artificial intelligence languages may be made complete; the overall design and suggested control structures of artificial intelligence languages provide new ideas for structuring mechanical theorem proving programs.

A fourth area of incompleteness relates to our solution of the incompleteness problem for the deductive system of artificial intelligence languages. This new problem concerns the use of models. Use of models in the form of either examples or abstractions of the actual problem is highly compatible with the problem reduction method which is the inference system of artificial intelligence languages. Use of models cannot be arbitrarily added to otherwise complete inference systems without possibly losing completeness.

The problem reduction method remains complete (within its domain of no negated goals) with a very general usage of models. There is substantially more difficulty in making its complete extension compatible with models.

We will present some examples which demonstrate within our

complete inference system the necessity for attempting to solve goals which are known to be unsolvable. We will also present modified inference procedures which are syntactically less restrictive but are more compatible with the use of models.

1.3. Control of theorem proving programs

Control is a critical issue in the design and use of present automatic theorem proving programs. Previous theorem proving systems are criticized in the next chapter for the lack of ability to control the expansion of the search space except usually by setting a few global parameters. The approach of PSTP is one proposed solution to this problem, with user programmed search strategies and filters ordering and pruning the search space.

Another alternative solution to the problem is suggested by the system of Henschen and Wos [19] which provides a theorem proving language to manipulate lists of clauses, directing resolutions be performed on, for example, the top clauses of the first and second lists. This allows for the easy programming of most standard resolution refinements.

Both systems are awkward for specifying certain kinds of strategy to the theorem prover.

PSTP's search order capability is completely general, although it may be difficult in any particular case to write the appropriate numerical evaluation function. (Given sufficient

numerical range, any sequence of tests may be represented by the sum of shifted values of the test.) It is furthermore limited to specification of search strategy; it has essentially no capability for altering the inference system being used. Thus, it is very difficult to change PSTP to use hyper-resolution rather than the linear MESON procedure.

Henschen's system is also limited in its ability to specify inference systems. His system would have similar difficulty to PSTP in changing from resolution to the linear MESON procedure since his system only understands clauses, not chains.

Some kinds of strategy information are difficult to provide to Henschen's system too. The basic unit manipulated by Henschen's system is the list of clauses, clauses at the beginning of the list being resolved upon, and newly generated clauses being added to the end of the list. Strategy is specified primarily by directing which clause lists are to be used presently. There are only a small number of clauses lists which may be active at any given time. In length preference strategies, separate lists would be maintained for clauses of length 1, 2, ... This can effectively be done in cases where the parameter (like length in this example) takes on only a small range of values. If the parameter has a much wider range of values, such as a size or complexity measure, Henschen's system will have more difficulty dealing with it given the restricted number of currently active clause lists.

Thus, one of the objectives of this thesis is to do the

preliminary work (i.e., investigating issues of completeness of artificial intelligence languages) not only to study these languages for artificial intelligence problem solving applications, but to consider application of these type of languages to the theorem proving task. The use of an artificial intelligence language with the required logical properties (a standard deduction system, and completeness, or an understanding of where completeness is lacking), represents an interesting alternative to PSTP and Henschen's theorem proving system for specifying strategy in attempting a proof. Although present artificial intelligence languages have several weaknesses in their logical characteristics which this thesis tries to point out and remedy, they have made significant contributions in the area of control, and it is these contributions which we hope to see used in future theorem proving systems.

1.4. Inconsistency and Uncertainty

The practical and epistemological suitability of formal logic (and specifically the first order predicate calculus) for performing the kinds of reasoning demanded of artificial intelligence programs has been extensively argued. Some principal objections to its use relate to formal properties restricting representation of inconsistency and uncertainty. We attempt to answer some of these objections here by showing that theorem proving programs are often more tolerant of irrelevant

inconsistency than is often believed, and that it is possible to develop a logic which extends both fuzzy logic (for representing uncertainty) and a four valued logic (for representing inconsistency.) We will not further develop the properties of this logic here, but this logic could be used as a basis for answering objections to more orthodox formal theories.

One of the objections to the use of mechanical theorem proving in natural reasoning is concerned with the handling of inconsistency in the data base.

It is alleged that, following the definitions of the underlying logical system, i.e., the first order predicate calculus in which any inconsistency results in the truth of everything, the presence of inconsistent data will result in the generation of proofs for questions unrelated to the inconsistency.

Even restricting our attention to conventional (first order predicate calculus) logic, this allegation is false in two respects.

First, the existence of an inconsistency in the data base is no guarantee that it will ever be discovered in reasonable time. If all inconsistencies were so readily discovered as to make use of data bases containing inconsistencies impossible, then the problem of creating efficient theorem provers would be solved as they themselves generate proofs by discovering inconsistencies. Briefly and informally, if partial deductions are generated in increasing order of their complexity, and if the complexity of

the discovery of the irrelevant inconsistency exceeds the complexity of the least complex proof of the theorem, then the inconsistency will not be discovered before the theorem is proved. Similarly, if (as is commonly true in a theorem prover based question answering system) the question is not a theorem, the inconsistency will still not be discovered if the cost of discovering it exceeds the resources assigned to answer the question.

Second, certain mechanizations of first order predicate calculus impose their own relevance criteria on items in the data base that prevent irrelevant inconsistencies from entering into an attempted proof. One of the most powerful, widely used, and intuitive refinements of resolution is the set of support strategy. Briefly, the set of support strategy customarily divides the set of clauses denoting the problem into two parts: the negated theorem (plus possibly special hypotheses) and the axioms (plus possibly special hypotheses). No resolutions are permitted among elements of the second set (the axioms; every resolvent must have as a parent a member of the first set or a descendant of a member of the first set. Although the proof of completeness of resolution using the set of support strategy prescribes that (for completeness to be maintained) the second set must be consistent, allowing it to be inconsistent is a sometimes useful generalization.

If there is no way for the theorem prover to bring the clauses participating in the irrelevant inconsistency into the search for

a proof of the theorem, then the inconsistency will not in any way affect the search.

For example, consider a data base consisting solely of the two assertions A and ¬A. According to the definition of the first order predicate calculus and the resolution theorem proving procedure without any refinements, every proposed theorem will have a proof. However, if the set of support strategy is used, only A and ¬A can be proved. If a proof is attempted for B, it will be discovered that ¬B (the negation of the theorem) cannot be resolved with any other clause. The resolution of A and -A is forbidden by the set of support strategy. Thus, no proof of B will be found.

Although this definition of relevance (related as it is to the failure to discover inconsistencies due to finding no set of literals linking clauses of the theorem to clauses in the inconsistency) is rather syntactic in form, it appears to be compatible with conventional notions of relevance. If a person held the inconsistent beliefs that it was both raining and not raining at some fixed point and time, he would be unlikely to let either belief interact with the determination of some other unrelated question such as the location of a pencil. The mechanical theorem proving system would behave similarly since there will not in general be any link in the data between issues of rain and pencil locations.

A mechanical theorem proving system using the set of support strategy in this way seems to deal with inconsistency in

substantially the same manner as a human being. If the inconsistency is a subtle one, the human will simply not notice it, and it is not likely to be discovered by a theorem prover since the cost of its discovery will exceed the cost of the cheapest proof or the resources allocated to discovering a nonexistent proof. Humans can successfully deal with even blatant inconsistencies in their beliefs by recognizing these beliefs are irrelevant to the question at hand. The set of support strategy similarly prescribes a relevance criterion which must be met for the inconsistent data to be used.

Although unrelated to the approach of this thesis, there is yet a third way in which formal theorem proving systems can deal with inconsistency. This way is to extend the logic being used to either classify facts into several levels representing degree of belief in them or to directly represent inconsistency in the logic.

The first of these possibilities is exemplified by Rescher [40]. Here, beliefs are classified into modal categories indicating strength of belief. If a belief system is inconsistent, it can be made consistent by stripping away beliefs by modal category in order of ascending degree of belief.

The second of these possibilities is exemplifed by Belnap [3,4]. Belnap's model is a question answering system which accepts inputs and answers queries. In contrast to Rescher's system in which one must effectively assign different credibility to different beliefs (placing them in different modal

categories), Belnap essentially assumes that we have no reason to judge the credibility of one input higher than that of another (possibly contradictory) input. His way of handling inconsistency is to extend the conventional two valued logic to a four valued logic. The four values are: **T** (indicating the system has been told the proposition is true), **F** (indicating the system has been told the proposition is false), **Both** (indicating the system has been told the proposition is true and the proposition is false), and **None** (indicating the system has neither been told the proposition is true nor the proposition is false). Belnap develops truth tables computing the value of compound expressions of propositions having these values (they are quite intuitive: for example, **Both** $\wedge$ **F** is **F**, **Both** $\vee$ **F** is **Both**, $\neg$**Both** is **Both**) and a mechanism for updating the state of the question answering system as new assertions are made.

Another formalism for dealing with uncertainty if not directly with inconsistency is the fuzzy logic of Zadeh [55]. Here, propositions are assigned truth values in the range [0,1] rather than {T,F}. An assignment of truth value 1 is essentially equivalent to an assignment of truth value T in conventional logic; an assignment of truth value 0 is essentially equivalent to an assignment of truth value F in conventional logic. In this way, fuzzy logic is an extension of the conventional two valued logic. Values between 0 and 1 represent further degrees of truth. The truth value of $\neg$A is 1 minus the truth value of A. Thus, fuzzy logic is not capable of dealing with A and $\neg$A both being asserted.

However, fuzzy logic can itself be extended to considering not just numbers in the range [0,1], but pairs of numbers in the range [0,1] as truth values. The first number x in the truth value <x,y> denotes the degree of truth or degree of confidence in the truth of the proposition to which it is assigned; the second number y denotes the degree of falsity or degree of confidence in the falsity of the proposition.

Truth values for non-atomic formulas are computed by the following rules:

$$\neg\langle x,y\rangle = \langle y,x\rangle$$

$$\langle x_1,y_1\rangle \vee \langle x_2,y_2\rangle = \langle \max(x_1,x_2), \min(y_1,y_2)\rangle$$

$$\langle x_1,y_1\rangle \wedge \langle x_2,y_2\rangle = \langle \min(x_1,x_2), \max(y_1,y_2)\rangle$$

If truth values are restricted to <1,0> and <0,1>, we have the conventional two valued logic in which <1,0> denotes **T** and <0,1> denotes **F**. If truth values are restricted to <x,1-x> $x \in [0,1]$, we have ordinary fuzzy logic. If truth values are restricted to <1,0>, <0,1>, <1,1>, and <0,0>, we have the four valued logic of Belnap in which <1,0> denotes **T**, <0,1> denotes **F**, <1,1> denotes **Both**, and <0,0> denotes **None**. If truth values may be freely taken from <x,y> $x,y \in [0,1]$, we have a new logic which is an extension of both fuzzy logic and Belnap's four valued logic. This last new logic will be equally capable of dealing with inconsistency as Belnap's four valued logic yet will have the additional facility of representing, for example, the credibility of information providers to the system by assigning differing numerical values to propositions.

An important point about this generalization of fuzzy logic and Belnap's four valued logic is that the theory for the latter also provides a theory for the new logic. The operations for computation of truth values and computation of alteration in the data base when assertions are made are based in theory on the truth values **T, F, Both, None** forming approximation and logical lattices. The new extended set of truth values $\langle x,y \rangle$ $x,y \in [0,1]$ also form lattices in precisely the same way, so that operations and their justification are analogous to those in Belnap's logic. These results will be presented in a separate paper.

2. An Inference System for Artificial Intelligence Languages

2.1. Introduction

This chapter is concerned with an inference system for artificial intelligence languages. Early sections of the chapter present terminology and background material on inference systems related to the one we propose as appropriate for artificial intelligence languages. In particular, Herbrand's theorem and resolution and model elimination procedures are described.

The MESON and linear MESON procedures are then described. These are similar to model elimination except for their more obvious relationship to the problem reduction method. The Programmable Strategy Theorem Prover (PSTP) theorem proving program using the linear MESON procedure is then described in substantial detail with experimental results.

The $_{\pi}$ theorem proving procedure with refinements for more efficient handling of the equality relation and use of models is then described. The $_{\pi}$ procedure is a complete theorem proving procedure which seems particularly well suited for use in artificial intelligence languages. It is essentially an abstraction of the MESON and linear MESON procedures which ignores representation issues and merges into a single problem reduction like operation the problem reduction operation and a form of reasoning by contradiction.

The inference system which is uniformly employed by artificial intelligence languages is essentially based on the problem reduction method. This however is an incomplete inference system for the propositional or first order predicate calculus since its handling of the negations of goals is incomplete, allowing for no proof by contradiction mechanism. We will present in this chapter a compatible extension of the presently used problem reduction method which is complete. This minimally alters the present inference system, requiring only that negated goals be asserted in the context in which their proofs are being attempted. This can easily be accomplished even in present artificial intelligence languages by including assertions of negated goals at the beginning of the code of consequent theorems.

Especially when, as here, completeness can be achieved so easily, we regard completeness as a desirable property. It assures us that if a proof cannot be found, it is not the fault of the inference system. Failure to find a proof must be a consequence of faulty axiomatization or incomplete other components of the language, such as pattern matching or search strategy. This is important since incompleteness, particularly incompleteness in the inference system, is usually demonstrable even with very simple (and thus more likely to occur) examples. Certainly the examples we show for the incompleteness of the problem reduction method, and for the first proposed model relative $_{\pi}$ procedure in the next chapter are all simple.

Besides the theoretical desirability of completeness, another reason for being concerned with completeness in the inference system is that some things can be done in different and possibly superior ways using a complete system rather than the currently used incomplete systems.

A good example of this is the generation of contingent plans. In QA4, a general schema for generating contingent plans is to assert in local contexts different values for undetermined predicates, construct a plan for each value, and synthesize a contingent plan from the resulting single plans.

Consider the QA4 example of trying to have fun. One can have fun at the beach if it is sunny, or can have fun at the movie if it is rainy (it is assumed it is either sunny or rainy).

This can be axiomatized as:

Example 2.1.

1. sunny → havefun(at-beach)
2. rainy → havefun(at-movie)
3. sunny ∨ rainy
4. havefun(s) → G

---

5. G

Here, we have introduced the global goal G rather than just using goal havefun(s) as the top goal. This is done for technical reasons so that the top goal is variable free.

The QA4 approach to solving this problem involves generating and combining separate plans for the two cases concerning whether it is rainy or sunny.

Representing these separate cases in MESON procedure proofs, we have

Example 2.2.

```
 1. sunny → havefun(at-beach)
 2. rainy → havefun(at-movie)
 3.1. sunny
 4. havefun(s) → G
 5. G                                          theorem
 6. [G] havefun(s)                             extend by 4
 7. [G] [havefun(at-beach)] sunny              extend by 1
 8. ∅                                          extend by 3.1
```

and

```
 1. sunny → havefun(at-beach)
 2. rainy → havefun(at-movie)
 3.2. rainy
 4. havefun(s) → G
 5. G                                          theorem
 6. [G] havefun(s)                             extend by 4
 7. [G] [havefun(at-movie)] rainy              extend by 2
 8. ∅                                          extend by 3.2
```

The synthesized plan is then "if it is sunny then go to the beach else go to the movie" (or, equivalently, "if it is rainy then go to the movie else go to the beach").

An alternative approach possible in the complete MESON procedure is demonstrated in the following proof:

```
 1. sunny → havefun(at-beach)
 2. rainy → havefun(at-movie)
 3. sunny ∨ rainy
 4. havefun(s) → G
 5. G                                          theorem
 6. [G] havefun(s)                             extend by 4
 7. [G] [havefun(at-beach)] sunny              extend by 1
 8. [G] [havefun(at-beach)] [sunny]
    ¬rainy                                     extend by 3a
 9. [G] [havefun(at-beach)] [sunny] [¬rainy]
    ¬havefun(at-movie)                         extend by contrapositive of 2
10. [G] [havefun(at-beach)] [sunny] [¬rainy]
    [¬havefun(at-movie)] ¬G                    extend by contrapositive of 4
11. ∅                                          reduce
```

A synthesized plan can now be generated from this proof in a fashion similar to the extraction of programs from resolution proofs developed by Waldinger [51]. Although there are

undoubtedly more direct approaches to generating the plan from the proof, that the plan can be derived is evident since the MESON procedure proof has an equivalent resolution proof to which the Waldinger algorithm may be applied.

2.2. Terminology

Definition 2.1. A term is defined to be
(1) a variable, or
(2) a function symbol followed by a list of terms. The term is called a constant if the list of terms following the function symbol is empty.

Variables will usually be denoted by lower case letters at the end of the alphabet: e.g., u,v,w,x,y,z. Constants will usually be denoted by lower case letters at the beginning of the alphabet: e.g., a,b,c,d,e. Function symbols of higher degree will usually be denoted by lower case letters occurring after constants in the alphabet: e.g., f,g,h,j,k. Example terms are then x, y, a, b, f(a,b), g(f(x,h(y))).

Definition 2.2. An atomic formula is defined to be a predicate symbol followed by a list of terms. The atomic formula is called a proposition if the list of terms following the predicate symbol is empty.

Predicate symbols will usually be denoted by upper case letters: e.g., F, P, Q, R. Example atomic formulas are then P,

Q(a,b), R(f(x,h(y))). The parentheses surrounding the list of terms following the predicate symbol and commas separating terms are often omitted: Qab, Rf(xh(y)).

Definition 2.3. A formula is defined to be

(1) an atomic formula,

(2) $\forall x B$ where B is a formula and x is a variable,

(3) $\exists x B$ where B is a formula and x is a variable,

(4) $\neg B$ where B is a formula,

(5) $(A \vee B)$ where A and B are formulas,

(6) $(A \wedge B)$ where A and B are formulas,

(7) $(A \rightarrow B)$ where A and B are formulas, or

(8) $(A \leftrightarrow B)$ where A and B are formulas.

Formulas without the parenthetical structure dictated by the above definition can be interpreted according to the following rules:

(1) the operators listed in decreasing order of precedence are $\forall x$, $\exists x$, $\neg$, $\vee$, $\wedge$, $\rightarrow$, and thus, for example, $-A \vee B \rightarrow C$ is equivalent to $((-A \vee B) \rightarrow C)$,

(2) $\vee$, $\wedge$, and $\leftrightarrow$ are associative and thus $A \vee B \vee C$ is equivalent to $(A \vee (B \vee C))$ and $((A \vee B) \vee C)$, $A \wedge B \wedge C$ is equivalent to $(A \wedge (B \wedge C))$ and $((A \wedge B) \wedge C)$, and $(A \leftrightarrow B \leftrightarrow C)$ is equivalent to $(A \leftrightarrow (B \leftrightarrow C))$ and $((A \leftrightarrow B) \leftrightarrow C)$, and

(3) $\rightarrow$ associates to the right and thus $A \rightarrow B \rightarrow C$ is equivalent to $(A \rightarrow (B \rightarrow C))$.

Formulas are intended to be evaluated to true or false. The value of $\forall xA$ is true if A is true for every value of x and false

otherwise. The value of $\exists x A$ is true if A is true for some value of x and false otherwise. The value of $\neg A$ is true if A is false and false if A is true. The value of $A \vee B$ is true if A or B is true and false if both are false. The value of $A \wedge B$ is true if A and B are true and false if either is false. The value of $A \rightarrow B$ is true if A is false or B is true and false if A is true and B is false. The value of A B is true if A and B are true or A and B are false and false if they are neither both true nor both false.

Definition 2.4. A literal is defined to be a formula of the form A or $\neg A$ where A is an atomic formula.

Definition 2.5. The complement $\sim C$ of the literal C is A if C is of the form $\neg A$, $\neg A$ if C is of the form A.

Definition 2.6. An assertion is defined to be a formula of the form $A_1 \wedge \ldots \wedge A_m \rightarrow C_1 \vee \ldots \vee C_n$ $(m \geq 0, n \geq 1)$ where each A_i and each C_j is a literal. Each A_i is called an antecedent of the assertion and each C_j is called a consequent of the assertion.

Definition 2.7. An implication is defined to be a formula of the form $A_1 \wedge \ldots \wedge A_m \rightarrow C$ $(m \geq 1)$ where each A_i and C is a literal, i.e., an implication is an assertion with n=1.

Definition 2.8. A disjunction is defined to be a formula of the form $C_1 \vee \ldots \vee C_n$ $(n \geq 1)$ where each C_j is a literal, i.e., a disjunction is an assertion with m=0. It is also called a clause.

Definition 2.9. A premise is defined to be a formula of the form C where C is a literal, i.e., a premise is an assertion with n=1 and m=0. It is also called a unit clause since it is a clause with a single literal.

Definition 2.10. A positive literal is an atomic formula.

Definition 2.11. A negative literal is a negated atomic formula.

Definition 2.12. A Horn clause is a clause containing not more than one positive literal.

Definition 2.13. A Horn set is a set of Horn clauses.

Definition 2.14. A positive clause is a clause containing only positive literals, i.e., having no negative literals.

Definition 2.15. A negative clause is a clause containing only negative literals, i.e., having no positive literals.

Definition 2.16. A mixed clause is a clause containing both positive and negative literals.

2.3. Problem reduction method

The problem reduction method is a widely understood, very general, and nearly universally applicable problem solving technique. Informally defined, the problem reduction method consists of finding for a goal conjunctive sets of subgoals the

simultaneous solution of one of these constituting a solution of the original goal. The method is recursively applied to each generated subgoal until the resultant subgoals are all immediately solvable, or some subgoal is unsolvable and that particular attempted solution is abandoned.

In this thesis, we are concerned with the solution of problems expressed symbolically. Accordingly, we will give a more formal definition of the problem reduction method as applied to symbolic expressions.

Let $B_1,\ldots,B_p$ be a set of assertions of the form $A_1 \wedge \ldots \wedge A_m \rightarrow C$ $(m \geq 0)$ (implications and premises), and C be a single literal goal. An implication B_k of the form $A_1 \wedge \ldots \wedge A_m \rightarrow C$ represents the statement that "to solve (prove) C, solve (prove) A_1, A_2, ..., A_m". A premise of the form C (i.e., m=0) represents that C is immediately solvable (provable, i.e., true).

Following is a definition of the problem reduction method in terms of its single inference rule.

Definition 2.17. $B_1,\ldots,B_p \vdash_p C$ if

B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C$ $(m \geq 0)$

and $B_1,\ldots,B_p \vdash_p A_1$

:

and $B_1,\ldots,B_p \vdash_p A_i$

:

and $B_1,\ldots,B_p \vdash_p A_m$ for some k $(1 \leq k \leq p)$.

Here, "$\vdash_p$" represents solvability (provability) by the problem reduction method.

It can be readily seen that the problem reduction method is complete in those instances where the negation of a goal has no meaning. In other words, if $B_1,\ldots,B_p \vdash C$ then $B_1,\ldots,B_p \vdash_p C$, i.e., C is derivable by the problem reduction method from $B_1,\ldots,B_p$. This completeness where negations of goals are meaningless is a result of the problem reduction method being isomorphic to the input resolution procedure, a complete theorem proving procedure for Horn formulas which are a representation for problems with meaningless negations.

However, the problem reduction method is incomplete when applied in situations where the negations of goals are employed. A simple example is the attempt to solve, by the problem reduction method, the goal A given implications $B \leftarrow A$ and $\neg B \leftarrow A$. It is apparent that, under the usual interpretation of $\neg$, that A is a consequence of $B \leftarrow A$ and $\neg B \leftarrow A$. Yet the problem reduction method has insufficient logical capability to detect this fact. It merely establishes separate, disjunctive subgoals B and $\neg B$, recognizes that it can solve neither, and fails.

A less serious logical difficulty of the problem reduction method is its incompleteness in the absence of contrapositives of implications. From B and $\neg A \leftarrow \neg B$ we can deduce A, but not by the problem reduction method. The contrapositive $B \leftarrow A$ of $\neg A \leftarrow \neg B$ must be included among the implications to generated a problem reduction method solution. This difficulty is also related to the negations of goals since, if negations of goals are meaningless, the contrapositives would also lack meaning since they contain negated goals.

We shall be concerned in this chapter with presenting a theorem proving (problem solving) procedure which addresses the logical difficulties of the problem reduction method while retaining its form.

The necessity for presenting contrapositives is retained in this theorem proving method.

The impossibility of proving A from $B \leftarrow A$ and $\neg B \leftarrow A$ by the problem reduction method is solved by modifying the single problem reduction method inference rule. Instead of using only originally supplied implications to generated subgoals, the negations of higher subgoals may also be used.

For example, in the problem cited above, we can establish B as a subgoal of A using $B \leftarrow A$. We can then establish $\neg A$ as a subgoal of B using $\neg A \leftarrow B$ (the contrapositive of $\neg B \leftarrow A$). At this point, the ordinary problem reduction method would fail since $\neg A$ occurs as the consequent of no implication. However, we permit the use of the negations of higher goals as premises. Thus, applying $\neg A$ (the negation of the top goal A) and acting as if $\neg A$ were an immediately solvable goal, we "solve" $\neg A$ which "solves" B which solves A.

The validity of this form of reasoning is demonstrated by showing it is a form of reasoning by contradiction: either A is true or false; if A is true, then the top goal A is true; if A is false, then (by $\neg A \leftarrow B$) B is true and (by $B \leftarrow A$) A is true, a contradiction; therefore, A must be true. Note that this reasoning makes no statement about the truth or falsity of B.

Given the implications $\neg A \rightarrow B$ and $B \rightarrow A$, A is true indepedently of the value of B.

2.4. Herbrand's theorem

In the propositional calculus (the restriction of the first order predicate calculus to atomic formulas which have 0-ary predicate symbols, and consequent elimination of quantifiers), it is easy to devise a proof procedure which can prove or disprove any formula. An example of such a proof procedure is Wang's algorithm [52] which, given any formula, will respond with either "VALID" or "NOT VALID" if the formula is or is not a theorem respectively, and will present a proof or disproof of the formula. In contrast to this nice situation in the propositional calculus, the first order predicate calculus is undecidable meaning that there is no procedure which can determine validity or invalidity of all formulas in finite time. Consequently, the strongest theorem proving procedure for the whole of the first order predicate calculus would have the following properties:
(1) it can prove, in a finite number of steps, any valid formula (completeness),
(2) it cannot prove any formula which is invalid (soundness), and
(3) an attempt to prove a formula which is invalid may result in an infinite computation.

Theorem proving procedures based on Herbrand's theorem have these properties. Prior to stating Herbrand's theorem, and

discussion of procedures based on it, it is necessary to present a series of definitions of standard forms of formulas referred to in the statement of Herbrand's theorem.

Definition 2.18. The prenex form of a formula is the logically equivalent formula in which all quantifiers appear at the beginning of the formula. More precisely, a formula A is said to be in prenex form if:

(1) it is in the form $Qx_1 \ldots Qx_n B$ where

(2) each Qx_i $(1 \leq i \leq n)$ is either $\forall x_i$ or $\exists x_i$ where x_i is a variable, and

(3) B is an open formula, i.e., B contains no quantifiers.

The list of quantifiers $Qx_1 \ldots Qx_n$ is called the prefix of A and B is called the matrix of A.

Example 2.3. (2) is the prenex form of (1).

(1) $\neg\forall y[Fyx \leftrightarrow \neg\exists z(Fyz \wedge Fzy)]$

(2) $\exists y\forall z\exists w[(Fyx \wedge Fyw \wedge Fwy) \vee (\neg Fyx \wedge (\neg Fyz \vee \neg Fzy))]$

Definition 2.19. The closure of a formula A in prenex form with prefix $Qx_1, \ldots, Qx_n$ and variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ is $\forall y_1 \ldots \forall y_m Qx_1 \ldots Qx_n B$ where B is the matrix of A. A closed formula in prenex form represents the standard form of a formula with respect to its quantifiers. (3) is the closure of (2), and is (1) in closed prenex form.

(3) $\forall x\exists y\forall z\exists w[(Fyx \wedge Fyw \wedge Fwy) \vee (\neg Fyx \wedge (\neg Fyz \vee \neg Fzy))]$

Definition 2.20. The Skolem functional form of a formula is a

special form in which all universal quantifiers have been eliminated. Universally quantified variables are replaced by "Skolem functions" of existentially quantified variables specially created for this purpose. Let A be a formula in closed prenex form. The Skolem functional form of A, denoted by A_H, is defined as follows:

(1) If A is existential (has only existential quantifiers) then A_H is just A.

(2) Otherwise, since A is in prenex form and must have at least one universal quantifier, it must be of the form $\exists x_1 \ldots \exists x_n \forall y B$ where B is a (not necessarily open) formula. A_H is then the Skolem functional form of A^* where A^* is defined to be $\exists x_1 \ldots \exists x_n B.\{y \leftarrow g(x_1,\ldots,x_n)\}$. $\{y \leftarrow g(x_1,\ldots,x_n)\}$ is a substitution which replaces all occurrences of y by $g(x_1,\ldots,x_n)$ where g is a specially created, unique n-ary Skolem function.

Thus the procedure for generating the Skolem functional form of A is to generate the finite sequence of formulas A, A^*, A^{**}, ... Each element of the sequence has one fewer universal quantifiers than its predecessor; the last element A_H is existential.

Example 2.4. These are the steps in the creation of the Skolem functional form of (3). (3.1) is $(3)^*$; (4) is $(3.1)^*$ or $(3)^{**}$ and is the Skolem functional form of (3).

(3.1) $\exists y \forall z \exists w[(Fya \wedge Fyw \wedge Fwy) \vee (\neg Fya \wedge (\neg Fyz \vee \neg Fzy))]$

(4) $\exists y \exists w[(Fya \wedge Fyw \wedge Fwy) \vee (\neg Fya \wedge (\neg Fyg(y) \vee \neg Fg(y)y))]$

The Skolem functions that were introduced are the 0-ary

function a which replaced x and the unary function g which replaced w.

Note that due to the non-uniqueness of the closed prenex form, the Skolem functional form is not generally unique. If, during the formation of the closed prenex form of the formula, one strives to place universal quantifiers as close to the beginning of the quantifier list as possible, the resulting Skolem functional form will look simpler, having Skolem functions with a smaller number of arguments.

It is also possible to generate the Skolem functional form of a formula without initially putting the formula into prenex form. This is discussed in Section 2.8.

Theorem 2.1. (Herbrand's theorem, version 1) The closed formula A in prenex form is a theorem if and only if a disjunction of ground instances of the matrix of A_H is a tautology.

To say that a formula is a tautology means that there is no assignment of truth values to ground instances of atoms for which the formula is false, i.e., the formula is invariably true under all possible truth value assignments.

Since most automatic theorem proving programs actually prove theorems by refuting their negations, the statement of Herbrand's theorem they use is based on the notion of unsatisfiability. A formula is unsatisfiable if there does not exists an assignment of truth values to ground instances of atoms in the formula for which the formula is true. In contrast to a tautology which is

always true, an unsatisfiable formula is always false. A truth functionally unsatisfiable formula is a ground formula which is unsatisfiable. Since a formula is a theorem if and only if its negation is unsatisfiable, Herbrand's theorem can now be stated as the following.

Theorem 2.2. (Herbrand's theorem, version 2) The closed formula A in prenex form is a theorem if and only if a conjunction of instances of the matrix of $\neg A_H$ (the negation of its Skolem functional form) is truth functionally unsatisfiable.

Note that the usage of $\neg A_H$ (the negation of the Skolem functional form) suggests an alternate method of preparing a formula for application of Herbrand's theorem. The formula may be negated first; then transformation to prenex form and substitution of Skolem functions for variables can be done. Since negating the formula changes universal quantifiers into existential quantifiers and vice versa, the substitution may be accomplished by replacing existentially (rather than universally) quantified variables with Skolem functions of universally quantified variables. For example, the Skolem functional form of the (already negated) formula $\forall x \exists y Pxy$ is $\forall x Pxg(x)$. The intuitive interpretation of the term g(x) is that it is the function providing for any x the y for which Pxy is true. Further, this approach more compellingly illustrates the fact that the final prefix always consists solely of universal quantifiers. Thus all variables in the matrix may be regarded as being implicitly universally quantified, and the prefix discarded. This way of

generating the input form for Herbrand's theorem is the one most used and described in theorem proving applications, although it differs from the historical approach described earlier.

In order to use the above definitions for proving theorems, it is necessary to define the set of terms which can be used in creating the needed instances. This set of terms is defined below.

Definition 2.21. The Herbrand universe for formula A consists of

(1) the variables of A_H, and

(2) terms constructed of n-ary function symbols of A_H followed by n terms of the Herbrand universe of A_H.

This definition of the Herbrand universe is actually an extremely important concept. It means essentially that all the vocabulary necessary to mechanically prove a theorem is contained in the statement of the theorem--there is no cause for resorting to the use of additional symbols.

Note that if A_H contains at least one function symbol with degree greater than zero, the Herbrand universe has an infinite number of elements. This precludes the possibility of enumerating a finite number of terms and testing the validity of a formula with each combination of values of (a finite number of) ground instances of the formula. This is the feature of the first order predicate calculus which may ultimately lead a theorem prover into an infinite computation when trying to prove a formula which is not a theorem.

The Herbrand universe of (4) is the set

{y,w,a,g(y),g(w),g(a),g(g(y)),...}.

So far in preparing the theorem to be proved for application of Herbrand's theorem, the quantifiers and variables have been manipulated, but the basic structure of the matrix has been left unchanged. Putting the matrix into a standard form will facilitate checking of truth functional unsatisfiability.

Definition 2.22. The conjunctive normal form of the matrix of a formula is the logically equivalent formula which is a conjunction of clauses.

The matrix of a formula is in conjunctive normal form if it has the form $(L_{11} \vee \ldots \vee L_{1n_1}) \wedge \ldots \wedge (L_{m1} \vee \ldots \vee L_{mn_m})$ where each L_{ij} is a literal. The matrix of any formula can be transformed into a (generally non-unique) conjunctive normal form.

Example 2.5. This is the generation of the conjunctive normal form of (4.1), the negation of (4). (4.2) and (4.3) are intermediate steps, and (5) is the final result of all preparation for applying Herbrand's theorem: the conjunctive normal form of the negation of the Skolem functional form of (1).

(4.1) $\neg\exists z\exists w[(Fya \wedge Fyw \wedge Fwy) \vee$
$(\neg Fya \wedge (\neg Fyg(y) \vee \neg Fg(y)y))]$

(4.2) $\forall z\forall w\neg[(Fya \wedge Fyw \wedge Fwy) \vee$
$(\neg Fya \wedge (\neg Fyg(y) \vee \neg Fg(y)y))]$

(4.3) $\forall z\forall w[(\neg Fya \vee \neg Fyw \vee \neg Fwy) \wedge$
$(Fya \vee (Fyg(y) \wedge Fg(y)y))]$

(5) $\forall z \forall w[(\neg Fya \vee \neg Fyw \vee \neg Fwy) \wedge$

$(Fya \vee Fyg(y)) \wedge$

$(Fya \vee Fg(y)y)]$

Besides being notationally convenient, putting $\neg A$ into conjunctive normal form has an extremely important advantage. This results from the fact that a conjunction is truth functionally unsatisfiable if and only if a (perhaps improper) subset of the set of conjuncts is truth functionally unsatisfiable. Thus, Herbrand's theorem can now be stated in the final form in which it is customarily used.

Theorem 2.3. (Herbrand's theorem, version 3) The closed formula A in prenex form is a theorem if and only if a conjunction of instances of the clauses of $\neg A_H$ is truth functionally unsatisfiable.

Example 2.6. This is a demonstration of the truth functional unsatisfiability of the conjunction of a set of instances of the clauses of (5). According to Herbrand's theorem, this constitutes a proof of (1).

I $\neg Faa$ is an instance of $\neg Fya \vee \neg Fyw \vee \neg Fwy$ obtained by replacing both y and w by a,

II $\neg Fag(a) \vee \neg Fg(a)a$ is also an instance of $\neg Fya \vee \neg Fyw \vee \neg Fwy$ obtained by replacing y by g(a) and w by a,

III $Faa \vee Fag(a)$ is an instance of $Fya \vee Fyg(y)$ obtained by replacing y by a, and

IV $Faa \vee Fg(a)a$ is an instance of $Fya \vee Fg(y)y$ obtained by replacing y by a.

That Faa must be false can be inferred from I; therefore, by LII and IV, Fag(a) and Fg(a)a must both be true. But under this assignment, clause II is assigned the value false, and the entire conjunction is assigned the value false. Thus, the conjunction of I, II, LII, and IV is unsatisfiable. The same result can be obtained by examining the truth table which considers all possible assignment of truth values to the atoms of clauses I, LI, III, and IV. The value of the conjunction of these four clause is tabulated below for each possible truth value assignment.

Faa	Fag(a)	Fg(a)a	I	LI	III	IV	I∧LI∧LII∧IV
T	T	T	F	F	T	T	F
T	T	F	F	T	T	T	F
T	F	T	F	T	T	T	F
T	F	F	F	T	T	T	F
F	T	T	T	F	T	T	F
F	T	F	T	T	T	F	F
F	F	T	T	T	F	T	F
F	F	F	T	T	F	F	F

The earliest applications of Herbrand's theorem to automatic theorem proving involved level saturation [42]. This consists of generating a sequence $C_0 \subset C_1 \subset \ldots C_n \subset \ldots$ of sets of instances of clauses in the matrix of the Skolem functional form of the formula to be proved. Each set is checked to determine if it is truth functionally unsatisfiable. If so, the procedure is terminated and the formula is a theorem. Otherwise, the next set in the sequence is generated. Should the next set be equal to the current set (no new clause instances were added) the formula is not a theorem. For most cases, however, (specifically, whenever there is at least one variable and one non-constant

function symbol), non-theorems will result in an infinite sequence of sets rather than this termination.

The criterion commonly used to determine which instances are to be generated as new instances of each set is the complexity of the instances. Thus C_n would contain all clauses with complexity measure less than or equal to n. The most common measure of complexity of a term is its function depth, a measure of the nesting of function symbols in the term. For a constant or variable, the function depth is defined to be zero. The function depth of a term composed of an n-ary function symbol (n>0) is defined to be one plus the maximum of the function depths of the arguments. For example, f(g(x)a) has function depth 2 and f(a) has function depth 1. Function depth can be easily computed by counting the maximum number of nested pairs of parentheses.

The rule for defining the set C_i could then be: C_i is defined to be the set of all instances of clauses in the matrix of the Skolem functional form of the formula to be proved formed by using terms of the Herbrand universe with function depth less than or equal to i in the substitutions.

The level saturation procedure takes its name from the fact that each successive set in the sequence represents a higher level of complexity of clause instances, and each set contains all instances of a specified complexity level.

This procedure is not at all practical for interesting problems due to the rapid growth rate in the number of clauses that must be generated and considered as the complexity level

increases. The procedure demands that all instances less complex than the most complex clauses required in the proof be generated, and the most complex clause required in the proof is frequently quite complex.

2.5. Resolution theorem proving procedure

The majority of automatic theorem proving programs for the first order predicate calculus are based on the resolution principle introduced by J. A. Robinson [41]. The resolution principle is fundamentally based on a rule of reasoning that the clause A ∨ B can be inferred from the conjunction of the clauses A ∨ C and A ∨ ¬C. A ∨ B is called a resolvent of the other two clauses which are called parent clauses of A ∨ B.

We will define resolution in terms of two operations: binary resolution and factorization.

2.5.1. Binary resolution. Let K and L be two clauses. Let K' and L' be variants of K and L respectively such that K' and L' have no variables in common. If C is a literal of K', D is a literal of L', and C_σ is $\neg D_\sigma$ or D_σ is $\neg C_\sigma$ for some unifying substitution σ then $(K'-C)_\sigma \vee (L'-D)_\sigma$ is a binary resolvent of K and L where (K'-C) is the clause K' with C removed and (L'-D) is the clause L' with D removed. This is called binary resolution in contrast to other resolution operations since exactly two literals are involved in the unification process: one from K and one from L.

We will show a schematic representation for various operations. The format will be the inputs followed by a horizontal line followed by the output. α, β, γ, and δ will denote sequences or disjunctions of literals; C and D will denote single literals. The resolution operation can be represented as:

$$\begin{array}{l} \text{input clause}_1 \ \alpha \vee C \vee \beta \\ \text{input clause}_2 \ \gamma \vee \neg C \vee \delta \\ \hline \text{resolvent } \alpha \vee \beta \vee \gamma \vee \delta. \end{array}$$

2.5.2. Binary factorization. Let K be a clause. If C and D are separate literals of K and $C\sigma$ is $D\sigma$ for some unifying substitution σ then $(K-D)\sigma$ is a binary factor of K. Similarly to binary resolution, this is called binary factorization since exactly two literals are involved in the unification process.

This can be expressed schematically as

$$\begin{array}{l} \text{input clause } \alpha \vee C \vee \beta \vee C \vee \gamma \\ \hline \text{factor } \alpha \vee C \vee \beta \vee \gamma. \end{array}$$

More general resolution and factorization operations can be defined. Factorization can be defined to permit an arbitrary subset of the literals of a clause to be unified together rather than just two. Resolution can be defined to permit an arbitrary subset of each clause to be unified against an arbitrary subset of the other clause. Each of these more general operations can be readily defined in terms of their binary counterparts.

2.5.3. Raw resolution. The derivation of the empty clause from a set of clauses by the resolution procedure constitutes a proof of the unsatisfiability of the set of clauses. We will call raw resolution the procedure which employs binary resolution

and factorization operations with no restrictions on the clauses they operate upon.

Example 2.7.

1.	¬Fya ¬Fyw ¬Fwy	
2.	Fya Fyg(y)	
3.	Fya Fg(y)y	
4.	¬Fya ¬Fay	factor 1(a,b)
5.	¬Faa	factor 4
6.	Fag(a)	resolve 2(a) and 5
7.	Fg(a)a	resolve 3(a) and 5
8.	¬Fag(a)	resolve 4(a) and 7
9.	□	resolve 6 and 8

Numerous refinements of raw resolution have been developed to reduce the number of possible inference operations that can be performed at any given point and thus reduce the number of derived clauses.

2.5.4. Tautology elimination. Tautology elimination permits the procedure to eliminate all clauses which contain two literals which are complementary to each other and are thus necessarily true under any assignment of truth values. Tautology elimination is compatible with most though not all refinements of resolution.

2.5.5. Subsumption. Clause C subsumes clause D if each literal of C_θ is a literal of D for some substitution θ. Since D is true whenever C is (C being more general), there is some justification for believing that D can be thrown away during the search for a proof. The subsumption refinement permits the elimination of newly generated clauses if a previously generated clause subsumes it (forward subsumption) and the elimination of previously generated clauses if newly generated clause subsumes it (backward subsumption). Subsumption in some form is

compatible with many refinements of resolution. Two major difficulties may occur in the arbitrary addition of the subsumption refinement to an otherwise incomplete procedure. The first concerns the compatibility of backward subsumption with the search strategy being employed. If a newly generated clause is a variant (identical except for variable renaming) of a previously generated clause and the latter is eliminated by backward subsumption, the search strategy may schedule inference operations to be performed (on the newly generated chain) in such a way that an infinite sequence of variants of the same clause may be generated, each variant eliminating its predecessor and having a lower priority for being operated upon. This problem can generally be overcome by always checking for forward subsumption before backward subsumption thereby always retaining the first rather than the most recent variant. The second major difficulty concerns the general incompatibility of subsumption with some refinements. For example, one may get into trouble using subsumption with linear refinements since the ancestry of a clause is important in determining what clauses it can be resolved against. Thus, one clause could subsume another with a different ancestry, yet the subsumed eliminated clause may be essential to a proof by virtue of its ancestry.

2.5.6. Set of support strategy. Let S be an unsatisfiable set of clauses and T be a subset of S such that S-T is satisfiable. The set of support refinement dictates that no members of S-T need be resolved against each other. Equivalently, the set of

support refinement dictates that every resolvent must have as an ancestor a member of T. This is a quite effective restriction on raw resolution. Generally, a theorem is posed in terms of a negated conclusion, some special hypotheses, and a set of axioms (e.g., axioms for groups or fields). The set T is often taken to be just the negated theorem or the negated theorem plus the special hypotheses. The set of axioms is assumed to be consistent, and no deductions are made from the set of axioms without referring somewhere in the deduction to clauses in the set of support, i.e., the negated theorem plus special hypotheses. This makes raw resolution somewhat more goal directed since without the set of support refinement, raw resolution (among other things) is free to draw all conclusions from the set of axioms without referring to the problem at hand.

2.5.7. Input resolution. Input resolution is an incomplete refinement of resolution in which one parent of each resolvent is an input clause.

Definition 2.23. An input resolution deduction of clause C from set of clauses S is a sequence of clauses $C_1,\ldots,C_n$ such that

(1) C_i $(1\leq i\leq m)$ is a member of S,

(2) C_i $(m<i\leq n)$ is a binary resolvent of a factor of C_{i-1} and a factor of C_j $(1\leq j\leq m)$, and

(3) C_n is C.

An input resolution refutation from set of clauses S is an input resolution deduction of the empty clause $\square$ from S.

Example 2.8. Following is an input resolution refutation from (5). Each operation in the proof including the formation of factors as required in the definition is explicitly shown.

1.	¬Fya ¬Fyw ¬Fwy	
2.	Fya Fyg(y)	
3.	Fya Fg(y)y	
[4.	¬Fya ¬Fay	factor 1(a,b)]
[5.	¬Faa	factor 4]
6.	Fg(a)a	resolve 3(a) and 5
7.	¬Fag(a)	resolve 6 and 4(a)
8.	Faa	resolve 7 and 2(b)
9.	□	resolve 8 and 5

2.5.8. Unit resolution. Unit resolution is an incomplete refinement of resolution in which one parent of each resolvent is a unit clause.

Definition 2.24. A unit resolution deduction of clause C from set of clauses S is a sequence of clauses $C_1,\ldots,C_n$ such that

(1) C_i $(1\leq i\leq m)$ is a member of S or a unit factor of a member of S,

(2) C_i $(m<i\leq n)$ is a binary resolvent of C_j $(1\leq j\leq i-1)$ and C_k $(1\leq k<j-1)$ where either C_j or C_k is a unit clause, and

(3) C_n is C.

A unit resolution refutation from set of clauses S is a unit resolution deduction of the empty clause □ from S.

Example 2.9. Following is a unit resolution refutation from (5).

1.	¬Fya ¬Fyw ¬Fwy	
2.	Fya Fyg(y)	
3.	Fya Fg(y)y	
4.	¬Faa	unit factor of 1
5.	Fag(a)	resolve 4 and 2(a)
6.	Fg(a)a	resolve 4 and 3(a)

7.	¬Fg(a)w ¬Fwg(a)	resolve 6 and 1(a)
8.	¬Fag(a)	resolve 7(a) and 6
9.	□	resolve 8 and 5

2.5.9. Linear resolution. Linear resolution is a complete refinement of resolution in which one parent of each resolvent is either an input clause or an ancestor of the other parent.

Definition 2.25. A linear resolution deduction of clause C from set of clauses S is a sequence of clauses $C_1,\ldots,C_n$ such that

(1) C_i $(1\leq i\leq m)$ is a member of S,

(2) C_i $(m<i\leq n)$ is a binary resolvent of a factor of C_{i-1} and a factor of C_j $(1\leq j<i-1)$, and

(3) C_n is C.

A linear resolution refutation from set of clauses S is an linear resolution deduction of the empty clause □ from S.

Example 2.10. Following is a linear resolution refutation from (5). Each operation in the proof including the formation of factors as required in the definition is explicitly shown.

1.	¬Fya ¬Fyw ¬Fwy	
2.	Fya Fyg(y)	
3.	Fya Fg(y)y	
4.	Faa ¬Fg(a)w ¬Fwg(a)	resolve 3(b) with 1(a)
5.	Faa ¬Fg(a)a Faa	resolve 4(c) with 2(b)
[6.	Faa ¬Fg(a)a	factor 5(a,c)]
7.	Faa Faa	resolve 6(b) with 3(b)
[8.	Faa	factor 7]
9.	¬Faa ¬Faa	resolve 8 with 1(b)
10.	¬Faa	resolve 9 with 8
11.	□	resolve 10 with 8

2.5.10. Paramodulation. Paramodulation [56] is an inference operation in addition to resolution designed to facilitate equality inferences.

Let K and L be two clauses. Let K' and L' be variants of K and L respectively such that K' and L' have no variables in common. If C is a literal of K', D is a literal of L', C (or D) is an equality literal of the form a=b or b=a, and the other literal contains a term unifiable with a for some unifying substitution σ, then $(K'-C)\sigma \vee (L'-D)\sigma \vee E\sigma$ where E is D (or C) with a single instance of the term unified with a replaced by b is a paramodulant of K and L.

This can be expressed schematically as

$$\begin{array}{l} \text{input clause}_1 \ \alpha \vee C \vee \beta \\ \text{input clause}_2 \ \gamma \vee D \vee \delta \\ \text{where C is a=b or b=a} \\ \hline \text{paramodulant } \alpha \vee \beta \vee \gamma \vee \delta \vee D' \\ \text{where D' is D with a single a replaced by b.} \end{array}$$

2.6. Model elimination theorem proving procedure

The model elimination theorem proving procedure was developed by Loveland [29]. The strong model elimination and model elimination with paramodulation refinements were also developed by Loveland in [30] and [31] respectively. We will take some liberties in presenting model elimination in slightly different fashion from Loveland.

In contrast to resolution, the fundamental element of a model elimination deduction is the chain. A chain is an ordered

sequence of literals. Two types of literals are distinguished: A-literals and B-literals. B-literals correspond to the literals present in clauses in resolution theorem proving. A-literals record ancestry information and represent assignments of truth values to literals already resolved upon. Model elimination is related to linear refinements of resolution, but since some ancestry information is retained in the chains it operates upon, model elimination is a complete input procedure in contrast to the incomplete input resolution operating on clauses rather than chains.

The input to the model elimination procedure as with resolution is a set of clauses. One of the clauses is designated to be the top chain of the refutation (the set of support) and is converted to a chain by merely selecting an ordering for its literals and fixing that order.

Definition 2.26. A model elimination (ME) deduction of chain K from set of clauses S is a sequence $K_0,\ldots,K_n$ of admissable chains such that

(1) K_0 is the top chain derived from a member of S,

(2) K_n is K, and

(3) K_i $(1\leq i\leq n)$ is derived from K_{i-1} by extension by a member of S, or by reduction.

A ME refutation is a ME deduction of the empty chain $\emptyset$.

2.6.1. Admissability. An admissable chain is a chain in which (1) no B-literal is followed by a complementary B-literal or A-literal unless they are separated by at least one A-literal, (2) no A-literal is followed by an identical B-literal or A-literal, (3) no A-literal is followed by a complementary A-literal, and (4) the last literal of the chain is not an A-literal.

A chain is termed preadmissable if it satisfies requirements (1), (2), and (3).

2.6.2. Extension. The extension operation takes an admissable chain K and an input clause C as its arguments. Let K' and C' be variable disjoint variants of K and C. If the last literal of K' is unifiable with the complement of a literal of C', the chain consisting of K' followed by the remaining literals of C' with unifying substitution applied is formed. Each literal of the derived chain descended from a literal of K' is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; each literal of the derived chain descended from a literal of C' is designated to be a B-literal. If the derived chain is preadmissable, the admissable chain with all terminal A-literals deleted is returned; no chain is returned if the derived chain is not preadmissable.

This can be expressed schematically as

$$\frac{\begin{array}{c}\text{input chain } \alpha\ C_i \\ \text{input clause } C_1 \vee \ldots \vee C_n\end{array}}{\text{derived chain } \alpha\ [C_i]\ C_1 \ldots C_{i-1}\ C_{i+1} \ldots C_n.}$$

2.6.3. Reduction. The reduction operation takes an admissable chain K as its argument. If the last literal of K is unifiable with the complement of a preceding A-literal of K, the chain consisting of K with the last literal deleted and with unifying substitution applied is formed. Each literal of the derived chain is designated to be the same type of literal as its ancestor. If the derived chain is preadmissable, the admissable chain with all terminal A-literals deleted is returned; no chain is returned if the derived chain is not preadmissable.

This can be expressed schematically as

$$\frac{\text{input chain } \alpha\ [C]\ \beta\ \sim C}{\text{derived chain } \alpha\ [C]\ \beta.}$$

ME is essentially input resolution with (1) ordered derived clauses, (2) retention of resolved upon literals as A-literals, (3) absence of the factorization operation, and (4) presence of the reduction operation. Unlike the input resolution procedure, ME is complete in spite of the absence of the factorization operation required by input resolution.

Example 2.11.

1.	¬Fya ¬Fyw ¬Fwy	
2.	Fya Fyg(y)	
3.	Fya Fg(y)y	
4.	Faa [Fg(a)a] ¬Fg(a)w ¬Fwg(a)	extend by 1(a)
5.	Faa [Fg(a)a] ¬Fg(a)a [¬Fag(a)] Faa	extend by 2(b)
6.	Faa [Fg(a)a] ¬Fg(a)a [¬Fag(a)] [Faa] ¬Faa ¬Faa	extend by 1(b)
7.	Faa [Fg(a)a] ¬Fg(a)a [¬Fag(a)] [Faa] ¬Faa	reduce (e,g)
8.	Faa [Fg(a)a] ¬Fg(a)a	reduce (e,f)
9.	Faa	reduce (b,c)
10.	[Faa] ¬Faa ¬Faa	extend by 1(b)
11.	[Faa] ¬Faa	reduce (a,c)
12.	∅	reduce

2.6.4. Strong model elimination. Addition of the equivalent of the factorization operation in the resolution procedure results in strong model elimination. Strong model elimination is so named not because it is necessarily more powerful (probably the contrary is true), but because the chain rejection criteria corresponding to the model elimination admissability requirement are stronger.

Definition 2.27. A strong model elimination (strong ME) deduction of chain K from set of clauses S is a sequence $K_0,\ldots,K_n$ of acceptable chains such that

(1) K_0 is the top chain derived from a member of S,

(2) K_n is K, and

(3) K_i $(1\leq i\leq n)$ is derived from K_{i-1} by extension by a member of S, by reduction, or by factorization.

A strong ME refutation is a strong ME deduction of the empty chain ϕ.

2.6.5. Acceptability. An acceptable chain is a chain in which

(1) no B-literal is followed by a complementary B-literal or A-literal,

(2) no A-literal is followed by an identical B-literal or A-literal,

(3) no A-literal is followed by a complementary A-literal,

(4) no B-literal is followed by an identical A-literal, and

(5) the last literal of the chain is not an A-literal.

A chain is termed preacceptable if it satisfies requirements (1), (2), (3), and (4).

2.6.6. Extension. The strong ME extension operation is the same as for ME except it operates on and produces acceptable rather than admissable chains.

2.6.7. Reduction. The strong ME reduction operation is the same as for ME except it operates on and produces acceptable rather than admissable chains.

2.6.8. Factorization. The factorization operation takes an acceptable chain K as its argument. If the last literal of K is unifiable with a preceding B-literal of K, the chain consisting of K with the last literal deleted and with unifying substitution applied is formed. Each literal of the derived chain is designated to be the same type of literal as its ancestor. If the derived chain is preacceptable, the acceptable chain with all terminal A-literals deleted is returned; no chain is returned if the derived chain is not preacceptable.

This can be expressed schematically as

$$\frac{\text{input chain } {}_{\alpha}\, C\, {}_{\beta}\, C}{\text{derived chain } {}_{\alpha}\, C\, {}_{\beta}.}$$

The addition of the factorization operation makes strong ME correspond more closely to input resolution. The chain acceptability criteria now permit tautology elimination (both before and after the second complementary B-literal has been extended upon), an operation not completeness preserving for ME (lacking the factorization operation).

Example 2.12.

1. ¬Fya ¬Fyw ¬Fwy
2. Fya Fyg(y)

3. Fya Fg(y)y
4. Faa [Fg(a)a] ¬Fg(a)w ¬Fwg(a) extend by 1(a)
5. Faa [Fg(a)a] ¬Fg(a)a [¬Fag(a)] Faa extend by 2(b)
6. Faa [Fg(a)a] ¬Fg(a)a factor (a,e)
7. Faa reduce (b,c)
8. [Faa] ¬Faa ¬Faa extend by 1(b)
9. [Faa] ¬Faa factor (b,c)
10. ∅ reduce

2.6.9. Model elimination with paramodulation. Two additional operations are available for handling the equality relation specially in a fashion similar to paramodulation.

Definition 2.28. A model elimination with paramodulation (ME-P) deduction of chain K from set of clauses S is a sequence $K_0,\ldots,K_n$ of permissable chains

(1) K_0 is the top chain derived from a member of S,

(2) K_n is K, and

(3) K_i $(1\leq i\leq n)$ is derived from K_{i-1} by extension by a member of S, by reduction, by factorization, by p-extension by a member of S, or by p-reduction.

A ME-P refutation is a ME-P deduction of the empty chain ∅.

2.6.10. Permissability. A permissable chain is a chain in which

(1) no A-literal is followed by a complementary A-literal,

(2) no B-literal is followed by an identical A-literal, and

(3) the last literal of the chain is not an A-literal. A chain is termed prepermissable if it satisfies requirements (1) and (2).

2.6.11. Extension. The ME-P extension operation is the same as for ME except it operates on and produces permissable rather than admissable chains.

2.6.12. Reduction. The ME-P reduction operation is the same as for ME except it operates on and produces permissable rather than admissable chains.

2.6.13. Factorization. The ME-P factorization operation is the same as for strong ME except it operates on and produces permissable rather than acceptable chains.

2.6.14. P-extension. The p-extension operation takes a permissable chain K and an input clause C as its arguments. Let K' and C' be variable disjoint variants of K and C.

(a) If a literal of C' is of the form a=b or b=a where a is unifiable with a term in the last literal of K', the chain consisting of K' followed by the remaining literals of C' followed by a copy of the last literal of K' with a single instance of a replaced by b with unifying substitution applied is formed. Each literal of the derived chain descended from a literal of K' is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; each literal of the derived chain descended from a literal of C' is designated to be a B-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-extension is called p-extension <u>from</u> an input clause.

(b) If the last literal of K' is a=b or b=a where a is unifiable

with a term in a literal of C', the chain consisting of K' followed by the remaining literals of C' followed by a copy of the literal containing the term unified with a with a single instance of that term replaced by b with unifying substitution applied is formed. Each literal of the derived chain descended from a literal of K' is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; each literal of the derived chain descended from a literal of C' is designated to be a B-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-extension is called p-extension <u>to</u> an input clause.

This can be expressed schematically as

(a)

input chain α C

input clause $C_1 \vee \ldots \vee C_n$
where C_i is a=b or b=a

---

derived chain α [C] $C_1 \ldots C_{i-1} C_{i+1} \ldots C_n$ C'
where C' is C with a single a replaced by b.

(b)

input chain α C

input clause $C_1 \vee \ldots \vee C_n$
where C is a=b or b=a

---

derived chain α [C] $C_1 \ldots C_{i-1} C_{i+1} \ldots C_n$ C_i'
where C_i' is C_i with a single a replaced by b.

2.6.15. P-reduction. The p-reduction operation takes an acceptable chain K as its argument.

(a) If the last literal of K contains a term unifiable with the term a where a preceding A-literal of K is of the form a=b or b=a, the chain consisting of K followed by a copy of the last

literal with a single instance of the term matching a replaced by b with unifying substitution applied is formed. Each literal of the derived chain descended from a literal of K is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-reduction is called p-reduction <u>from</u> an A-literal.

(b) If the last literal of K is of the form a=b or b=a where a is unifiable with a term in a preceding A-literal of K or the last literal of K itself, the chain consisting of K followed by a copy of the preceding A-literal or last literal with a single instance of the term matching a replaced by b with unifying substitution applied is formed. Each literal of the derived chain descended from a literal of K is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-reduction is called p-reduction <u>to</u> an A-literal or self.

This can be expressed schematically as

(a)

input chain $_{\alpha}$ [D] $_{\beta}$ C
where D is a=b or b=a

---

derived chain $_{\alpha}$ [D] $_{\beta}$ [C] C'
where C' is C with a single a replaced by b.

(b)

input chain $_{\alpha}$ [C] $_{\beta}$ D
where D is a=b or b=a

---

derived chain $_{\alpha}$ [C] $_{\beta}$ [D] C'
where C' is C with a single a replaced by b.

C and D may be the same literal in p-reduction to self.

2.7. MESON theorem proving procedure

The MESON procedures were first reported on in [32]. These procedures are complete extensions motivated by the model elimination theorem proving procedure of the problem reduction method operating on AND/OR goal trees (MESON stands for "model elimination subgoal oriented). Here, they are presented for propositional (variable free) problems only with an occasional reference to the requirements of the procedures utilizing variables.

The procedures represent syntactic systems for adding to a goal tree information about goal-subgoal relationships and establishment of goals. The procedures return success or failure according to whether the top goal can be established or not respectively. Of course, the ability to return failure disappears when substitution is allowed, e.g., first order formulations. A returned value of failure for a problem

indicates either the top goal does not follow from the assertions or the search ordering and goal generation and deletion strategies specified by the planning routine are inadequate for the problem. (It is possible to write a complete planning routine which theoretically always returns success for solvable problems.)

We will now present two MESON procedures for goal tree analysis incorporating the new rules discussed above. The procedures are logically divided into four subprocedures with labels "initialize", "loop", "update_marks", and "update_goals".

The instructions placed at the label "initialize" define GOALS (the set of goals to be attempted) to be the set consisting of only the top goal and also initialize the goal tree.

The instructions placed at the label "loop" select a goal G from GOALS, an operation to be performed, and an assertion D if needed. The selected operation is then performed for the goal G and assertion D. Those operations try to establish goals or create subgoals.

The instructions placed at the label "update_marks" add newly generated subgoals to the tree and GOALS provided certain acceptance criteria are met.

The selection of the next goal in GOALS to be operated upon and the selection of the operation and the assertion to be used in operating on that goal are assumed to be accomplished by some externally specified planning routine ("the planner"). The planner, in addition to specifying a search strategy, may

restrict or totally eliminate use of some operations. For example, traditional goal tree procedures without a reasoning by contradiction mechanism correspond to a planner which never uses the operation at "op3".

The planner, by applying the operation at "op5" to a goal, removes the goal from GOALS and thereby signifies that no more operations will be applied to the goal.

If one wishes to insure completeness, the planner must in some order process all operations (except the operation at "op5") for each goal and potentially applicable assertion. The planner may select the goals of a conjunctive set of goals in any desired order to attempt their establishment. The procedures make no assumption as to whether the search is depth-first, breadth-first, or some mixture of these.

2.7.1. MESON procedure.

MESON procedure

initialize: Let GOALS be a set consisting of only the top goal. Initialize the goal tree to the top goal.

loop: If GOALS is empty, exit procedure with failure. Otherwise, let G be a goal in GOALS selected by the planner. The planner selects one of the following operations to be performed on G and selects D, a premise, implication, or general contrapositive of implication, as required by the operation.

op1: If G matches the premise D, mark G established and go to update_marks. Otherwise go to loop.

op2: If G matches the consequent of D, where D is an implication or general contrapositive of implication, let A be the set of antecedents of D and go to update_goals. Otherwise go to loop.

op3: If G matches the complement of an ancestor of G,

mark G contradicted and go to update_marks. Otherwise go to loop.

op4: If G matches an unexpanded partner of G not marked displaced or an unexpanded ancestor partner of G, mark G displaced and go to update_marks. Otherwise go to loop.

op5: Delete G from GOALS and go to loop.

update_marks: If G is top goal, exit procedure with success. If all partner goals of G are marked established, contradicted, or displaced, let G_1 be the parent of G, set $G=G_1$, mark G established and go to update_marks. Otherwise go to loop.

update_goals:

test1: If a member of A is identical to G or an ancestor of G, go to loop.

test2: If a member of A is complementary to another member of A, an unexpanded partner of G, or an unexpanded ancestor partner of G, go to loop.

Otherwise add the member of A to GOALS and to the goal tree as a conjunctive set of successors of G and go to loop.

2.7.2. MESON procedure for equality. The MESON procedure for equality incorporates rules for handling the equality relation. It differs from the MESON procedure in that three new operations are added. Also, the rules for disregarding newly generated subgoals (at "test1" and "test2") have not been proven to preserve completeness although we believe completeness is preserved with these rules applied. We maintain the update_goals subprocedure in the MESON procedure in the MESON procedure for equality with the admonition that if completeness is to be preserved these rules should be bypassed (at present).

For technical reasons, it is necessary to put in premises of

the form a=a for eacn term a or, if in a setting using free variables and substitutions, one must put in x=x and $f(x_1,\ldots,x_n)=f(x_1,\ldots,x_n)$ for each n-ary function symbol f. Such axioms can be replaced by appropriate procedure rules if desired.

MESON procedure for equality

initialize: (same as for MESON procedure)

loop: (preface and operations 1-5 same as for MESON procedure; only change is the addition of the following operations)

op6: If G contains a term matching term a where a=b or b=a is the consequent of D, where D is a premise, implication, or general contrapositive of implication, let A be the set consisting of G with a single instance of a replaced by b plus the antecedents of D and go to update_goals. Otherwise go to loop.

op7: If the consequent of D, where D is a premise, implication, or general contrapositive of implication, contains a term matching a where G is $a\neq b$ or $b\neq a$, let A be the set consisting of the complemented consequent of D with a single instance of a replaced by b plus the antecedents of D and go to update_goals. Otherwise go to loop.

op8: If H is an ancestor of G or G itself and H (resp. G) contains a term matching a where G (resp. H) is $a\neq b$ or $b\neq a$, let A be the set consisting of H (resp. G) with a single instance of a replaced by b and go to update_goals. Otherwise go to loop.

update_marks: (same as for MESON procedure)

update_goals: (same as for MESON procedure)

2.8. The Linear MESON theorem proving procedure

The linear MESON procedure is a variant of the model

elimination theorem proving procedure in which (1) each literal of the top chain and the derived chains of a deduction is replaced by its complement and (2) implications as well as disjunctions are permitted as axioms. It is also the linear form of the MESON procedure specified for goal-subgoal trees. Advantages of the linear MESON procedure are (1) the linear MESON procedure, though logically equivalent to model elimination, has the form of an affirmation rather than a refutation procedure and its proofs have a very natural interpretation in terms of goal-subgoal trees and (2) the linear MESON procedure has greater expressive power than alternative procedures in the potential use of implications rather than disjunctions to restrict application of inference operations to certain literals of axioms or (by replacing a disjunction by more than one implication) to facilitate differential treatment of the various implicative forms of axioms during the search for a solution (this capability is shared by model elimination in which each length n input clause generates n auxiliary chains only the last literal of which can be matched in inference operations).

In a theorem proving program permitting interaction between human user and mechanical proof procedure, it is desirable that as human-oriented a procedure as possible be employed. While it is correctly argued that all resolution type theorem proving procedures are machine-oriented and unsuited to extensive human computation, it is our contention that the linear MESON procedure is more human-oriented than other resolution related procedures.

This is a direct consequence of the relationship of the linear MESON procedure with the problem-reduction method. The linear MESON procedure is an extension of the problem-reduction method which is complete for the first order predicate calculus with equality. It augments the problem-reduction method by inference operations which perform reasoning by contradiction (reduction), which eliminate duplicate subgoals (factorization), and which deal with the equality relation (p-extension, p-reduction).

The linear MESON procedure represents the state of a search for solution as a set of chains. Each chain represents a subtree of the search space. The solution of all the subgoals represented in a chain constitutes a solution of the top goal. Different chains represent different alternative partial solutions of the top goal.

More specific claims of the linear MESON procedure being human-oriented are (1) it is an affirmation rather than a refutation procedure, (2) in keeping literals ordered in a chain, it automatically prevents (in goal-subgoal tree terms) the start of an attempt to solve another subgoal in the chain until the current one is solved, and (3) it is a procedure which remains complete if only input deductions are used. An input deduction is a deduction in which each element of the deduction (a linear MESON procedure chain) is derived by an inference operation applied to its predecessor or its predecessor and an (input) axiom.

In combination, the last two items permit the user to focus

his attention on a much smaller subset of the available data than is possible for many resolution based procedures.

In preparing a problem for input to the linear MESON procedure, it is first Skolemized to eliminate all quantifiers. This can be accomplished by the following procedure.

Let A be a formula using only the $\wedge$, $\vee$, $\rightarrow$, and $\neg$ logical connectives. Occurrences of $B \leftrightarrow C$ can, of course, be replaced by $B \rightarrow C \wedge C \rightarrow B$ or $B \wedge C \vee \neg B \wedge \neg C$, etc., to so restrict connective usage.

Let A be further restricted such that no quantifier quantifies a free variable of A or any variable already quantified at that point i.e., the scopes of variables with the same name are disjoint. A renaming of variables can guarantee this condition.

A positive occurrence of a quantifier in A is a quantifier in the scope of an even number (possibly zero) of negations, either explicit (in the case of the $\neg$ connective) or implicit (in the case of occurring in an antecedent of an implication).

The formula is Skolemized by (1) replacing each variable quantified by a positively occurring universal quantifier or a negatively occurring existential quantifier by a unique Skolem function whose arguments are the variables of all positively occurring existential quantifiers or negatively occurring universal quantifiers in whose scope the quantifier of the variable being replaced appears and (2) deleting all quantifiers from the formula.

The resulting quantifier free formula is then converted to a conjunction of assertions implying a conclusion. An assertion is

a (possibly empty) conjunction of literals (antecedents) implying a disjunction of literals (consequents). Note that if there are no antecedents in the assertion, the assertion is just a disjunction of literals (disjuncts), i.e., a clause. A conclusion is a conjunction of literals. Schematically, the transformed formula is in the form

$$((A_1^1 \wedge \ldots \wedge A_{m_1}^1 \rightarrow C_1^1 \vee \ldots \vee C_{n_1}^1) \wedge \ldots \wedge (A_1^p \wedge \ldots \wedge A_{m_p}^p \rightarrow C_1^p \vee \ldots \vee C_{n_p}^p) \rightarrow (G_1 \wedge \ldots \wedge G_q))$$

where A, C and G denote literals, and $p \geq 1$, $q \geq 1$, each $m_1 \geq 0$, each $n_j \geq 1$.

Example 2.13. The problem is: if a is a prime number and a times the square of some number u is b then a divides b. The initial formula is:

```
   ∀x∀y∀z∀w (prime(x) ∧ y*z=w ∧ divides(x,w)
                 → divides(x,y) ∨ divides(x,z))
 ∧ ∀x x*x=square(x)
 ∧ ∀x∀y∀z (x*y=z → y*x=z)
 ∧ ∀x∀y∀z (x*y=z → divides(x,z))
 ∧ ∃u a*square(u)=square(b)
 ∧ prime(a)
→ divides(a,b)
```

The Skolemized form ready for input to the linear MESON procedure is:

```
   (prime(x) ∧ y*z=w ∧ divides(x,w) → divides(x,y) ∨ divides(x,z))
 ∧ x*x=square(x)
 ∧ (x*y=z → y*x=z)
 ∧ (x*y=z → divides(x,z))
 ∧ a*square(c)=square(b)
 ∧ prime(a)
→ divides(a,b)
```

Problems with equality (as above) can be introduced without the need for specifying the symmetry, transitivity, and

substitutivity axioms if the special equality inference operations (p-extension, p-reduction) are used. The equality reflexive (x=x) and functionally reflexive axioms (e.g., square(x)=square(x), x*y=x*y) theoretically are required. The latter are not present in the above example since no special equality rules are required for the problem's solution.

By virtue of its derivation from model elimination, the linear MESON procedure is complete (given that a compatible set of inference and postprocessing operations are used) provided (1) the set of assertions is consistent (this requirement is equivalent to the requirement that the top chain of a model elimination deduction be in the minimally unsatisfiable set of input clauses) and (2) either the disjunctive axiom form is used or all implicative forms of each assertion are included among the assertions. The first condition can be eliminated by the addition of a special contradiction mechanism defined for the MESON procedure but not included in this formulation of the linear MESON procedure which permits the proof of any conclusion from an inconsistent set of assertions. The second condition can sometimes be eliminated in practice since it is often clear from the problem structure (as in the case of Horn formulas) that use of a subset of the implicative forms results in the possible deduction of all the chains that the disjunctive form would. Also, although the resulting procedure is, in general, incomplete it is sometimes desirable to restrict the search for a proof by not presenting the procedure with all the implicative forms of the assertions.

The fundamental element of a linear MESON procedure deduction is the chain. A chain is an ordered sequence of literals. Two types of literals are distinguished: A-literals and B-literals. B-literals correspond to the literals present in clauses in resolution theorem proving. A-literals record ancestry information and represent (in goal-subgoal tree terms) higher goals. All the literals in the theorem and axioms are B-literals. An A-literal is created in a newly derived chain from a B-literal in the parent chain when a set of literals whose conjunction implies the A-literal (a set of subgoals whose solution constitutes a solution to the goal represented by the A-literal) is added.

Definition 2.29. A linear MESON procedure deduction of chain K from problem P is a sequence $K_0,\ldots,K_n$ of acceptable chains such that

(1) K_0 is the conclusion of P,

(2) K_n is K,

(3) K_i ($1 \leq i \leq n$) is derived from K_{i-1} by extension (by an assertion C_i), factorization, reduction, p-extension (by an assertion C_i) or p-reduction, where C_i is an assertion of P or a lemma (see lemma formation operation below).

A linear MESON procedure solution of P (or proof of the conclusion of P) is a linear MESON procedure deduction of the empty chain ϕ. In general, for the inference system to be complete, the negation of the conclusion must be included among the assertions. The definitions of acceptable chains and the inference operations are given below.

2.8.1. Matching. If the two arguments to the matching procedure are terms, the matching procedure returns the most general unifier of the terms. If the two terms are not unifiable, the matching procedure fails.

If the two arguments to the matching procedure are literals and are both positive (unnegated) or both negative (negated) literals with unifiable atomic formulas, the matching procedure returns the most general unifier of the atomic formulas. If the two arguments are not both positive or both negative or the atomic formulas are not unifiable, the matching procedure fails.

2.8.2. Extension. The extension operation takes an acceptable chain K and an assertion C as its arguments. Let K' and C' be variable disjoint variants of K and C. If the last literal of K' matches a consequent of C', the chain consisting of K' followed by the antecedents and the complements of the remaining consequents of C' with matching substitution applied can be inferred. Each literal of the derived chain descended from a literal of K' is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; each literal of the derived chain descended from a literal of C' is designated to be a B-literal.

This can be expressed schematically as

$$\frac{\begin{array}{c}\text{input chain } \alpha\ C_i \\ \text{input assertion } A_1 \wedge \ldots \wedge A_m \rightarrow C_1 \vee \ldots \vee C_n\end{array}}{\text{derived chain } \alpha\ [C_i]\ A_1 \ldots A_m \sim C_1 \ldots \sim C_{i-1}\ \sim C_{i+1} \ldots \sim C_n.}$$

2.8.3. Factorization. The factorization operation takes an acceptable chain K as its argument. If the last literal of K matches a preceding B-literal of K, the chain consisting of K with the last literal removed and with matching substitution applied can be inferred. Each literal of the derived chain is designated to be the same type as its ancestor.

This can be expressed schematically as

$$\frac{\text{input chain } \alpha\ C\ \beta\ C}{\text{derived chain } \alpha\ C\ \beta.}$$

2.8.4. Reduction. The reduction operation takes an acceptable chain K as its argument. If the last literal of K matches the complement of a preceding A-literal of K, the chain consisting of K with the last literal removed and with matching substitution applied can be inferred. Each literal of the derived chain is designated to be the same type as its ancestor.

This can be expressed schematically as

$$\frac{\text{input chain } \alpha\ [C]\ \beta\ \sim C}{\text{derived chain } \alpha\ [C]\ \beta.}$$

2.8.5. P-extension. The p-extension ("p-" for paramodulation) operation takes an acceptable chain K and an assertion C as its arguments. Let K' and C' be variable disjoint variants of K and C. (a) If a consequent of C' is of the form a=b or b=a where a matches a term in the last literal of K', the chain consisting of K' followed by the antecedents and the complements of the remaining consequents of C' followed by a copy of the last literal of K' with a single instance of the term matching a

replaced by b with matching substitution applied can be inferred. Each literal of the derived chain descended from a literal of K' is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; each literal of the derived chain descended from a literal of C' is designated to be a B-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-extension is called p-extension from an assertion. (b) If the last literal of K' is of the form a≠b or b≠a where a matches a term in a consequent of C', the chain consisting of K' followed by the antecedents and the complements of the remaining consequents of C' followed by a copy of the complement of the consequent containing the term matching a with a single instance of that term replaced by b with matching substitution applied can be inferred. Each literal of the derived chain descended from a literal of K' is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; each literal of the derived chain descended from a literal of C' is designated to be a B-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-extension is called p-extension to an assertion.

This can be expressed schematically as

(a)

input chain $_{\alpha}$ C

assertion $A_1 \wedge \ldots \wedge A_m \rightarrow C_1 \vee \ldots \vee C_n$

where C_i is a=b or b=a

---

derived chain $_{\alpha}$ [C] $A_1 \ldots A_m \sim C_1 \ldots \sim C_{i-1} \sim C_{i+1} \ldots \sim C_n$ C'

where C' is C with a single a replaced by b.

(b)

input chain $_{\alpha}$ C

assertion $A_1 \wedge \ldots \wedge A_m \rightarrow C_1 \vee \ldots \vee C_n$

where C is a=b or b=a

---

derived chain $_{\alpha}$ [C] $A_1 \ldots A_m \sim C_1 \ldots \sim C_{i-1} \sim C_{i+1} \ldots \sim C_n \sim C_i'$

where C_i' is C_i with a single a replaced by b.

2.8.6. P-reduction. The p-reduction ("p-" for paramodulation) operation takes an acceptable chain K as its argument. (a) If the last literal of K contains a term matching the term a where a preceding A-literal of K is of the form a≠b or b≠a, the chain consisting of K followed by a copy of the last literal with a single instance of the term matching a replaced by b with matching substitution applied can be inferred. Each literal of the derived chain descended from a literal of K is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-reduction is called p-reduction <u>from</u> an A-literal. (b) If the last literal of K is of the form a≠b or b≠a where a matches a term in a preceding A-literal of K or the last literal of K itself, the chain consisting of K followed by a copy of the preceding A-literal or last literal with a single instance of the term matching a replaced by b with matching substitution applied can be inferred.

Each literal of the derived chain descended from a literal of K is designated to be the same type of literal as its ancestor except the last which is designated to be an A-literal; the last literal of the derived chain (in which an instance of a term matching a was replaced by b) is designated to be a B-literal. This form of p-reduction is called p-reduction to an A-literal or self.

This can be expressed schematically as

(a)
input chain α [D] β C
where D is a=b or b=a

---

derived chain α [D] β [C] C'
where C' is C with a single a replaced by b.

(b)
input chain α [C] β D
where D is a=b or b=a

---

derived chain α [C] β [D] C'
where C' is C with a single a replaced by b.

C and D may be the same literal in p-reduction to self.

If the parent chain K in the p-extension or p-reduction operation is itself derived by p-extension or p-reduction, the created A-literal in the derived chain may optionally be omitted with completeness unaffected (see [31]). There is a tradeoff here. If the A-literal is omitted, the derived chain is shorter and easier to read, and some future possible reductions and p-reductions may be eliminated. On the other hand, especially if the postprocessing operation specifies rejection of chains containing an A-literal followed by an identical A-literal or B-literal, retention of the A-literal may result in rejection of

more chains as being unacceptable. For example, this could prevent repeated p-extension by a=b from creating an endless sequence of chains ending alternately in Pa or Pb.

2.8.7. Postprocessing. A postprocessing operation takes a chain K (output from the extension, factorization, reduction, p-extension or p-reduction operation) as its argument and either rejects K as being non-acceptable and thus unusable as input to any inference operation or transforms the chain into an acceptable chain. Many different postprocessing operations can be written with different effects regarding efficiency and completeness. Four postprocessing operations are described in the following table. The table expresses possible relationships between each pair of literals in the chain. All the actions corresponding to true conditions are to be performed on the chain, except, of course, that if the action is to reject the chain then no other conditions need be checked or actions need be performed.

	Postprocessing operations			
	STRONG-SAVE	STRONG-DELETE	WEAK-SAVE	WEAK-DELETE
Condition	Action	Action	Action	Action
A-literal followed by identical A-literal	reject chain	reject chain	reject chain	reject chain
A-literal followed by complementary A-literal	reject chain	reject chain	reject chain	reject chain
A-literal followed by identical B-literal	reject chain	reject chain	reject chain	reject chain
A-literal followed by complementary B-literal		delete following B-literal		delete following B-literal
B-literal followed by identical A-literal	reject chain	reject chain		
B-literal followed by complementary A-literal	reject chain	reject chain		
B-literal followed by identical B-literal		delete following B-literal		delete following B-literal
B-literal followed by complementary B-literal	reject chain	reject chain		

If all the actions of the postprocessing operation specified in the table have been performed and the chain is not rejected, all terminal A-literals of the chain are deleted. This terminal A-literal deletion is called contraction.

The deletion action associated with the "B-literal followed by identical B-literal" condition is called ground factorization since it represents an instantiation-free usage of a generalized factorization operation which can delete non-terminal B-literals. Similarly, the deletion action associated with the "A-literal followed by complementary B-literal" condition is called

ground_reduction since it represents an instantiation-free usage of a generalized reduction operation which can delete non-terminal B-literals.

Several additional conditions and actions can be used in postprocessing operations such as the following which are available in PSTP but were not used in the present study: (1) rejecting chains containing a non-terminal A-literal that is an instance of a unit axiom (single literal input assertion), (2) rejecting chains containing a non-terminal A-literal that is an instance of a unit lemma (single literal derived assertion, see lemma formation operation below) by which the ancestor chain could have been extended when the A-literal was created, (3) deletion of B-literals which are instances of unit axioms, (4) deletion of B-literals which are instances of unit lemmas, (5) removal of all literals including and following the A-literal in the case of a B-literal followed by an identical A-literal (this is called factorization_truncation), (6) removal of all literals including and following the second A-literal in the case of an A-literal followed by a complementary A-literal (this is called reduction_truncation). All of these actions can be shown to preserve completeness.

For problems not involving the equality relation, the WEAK-SAVE and WEAK-DELETE postprocessing operations yield a complete inference system when the extension and reduction operations are used. The WEAK postprocessing operations correspond closely to the chain admissability criteria for model

elimination of [29]. Also for problems not involving the equality relation, the STRONG-SAVE and STRONG-DELETE postprocessing operations yield a complete inference system provided the factorization operation is used in addition to extension and reduction. The STRONG postprocessing operations correspond closely to the chain acceptability criteria for strong model elimination of [30]. Note that with these postprocessing operations, if the conclusion is variable free, its negation need not be included among the assertions since extension by the negation of the conclusion in any deduction from the conclusion would result in a non-acceptable chain with an A-literal followed by an identical B-literal or an A-literal followed by a complementary A-literal.

For problems involving the equality relation, a postprocessing operation which rejects a chain only if it has an A-literal followed by a complementary A-literal yields a complete inference system when the extension, reduction, p-extension and p-reduction operations are used. Also for problems involving the equality relation, a postprocessing operation which rejects a chain only if it has a A-literal followed by a complementary A-literal or a B-literal followed by an identical A-literal yields a complete inference system provided the factorization operation is also used. This corresponds closely to the chain permissability criteria for model elimination with paramodulation of [31]. We believe (but have no proof) that for problems involving the equality relation, the WEAK-SAVE and WEAK-DELETE postprocessing

operations yield a complete inference system when the extension, reduction, p-extension and p-reduction operations are used and the STRONG-SAVE and STRONG-DELETE postprocessing operations yield a complete inference system when the factorization operation is also used. The equality reflexive (x=x) and functionally reflexive axioms (e.g., f(x,y)=f(x,y)) theoretically are required in any case.

2.8.8. Lemma formation. An additional inference operation is used to create new assertions during contraction. A new assertion (called a lemma) consisting of the disjunction of the terminal A-literal being removed and all preceding A-literals whose scope (see below) exceeds the number of A-literals between them and the terminal A-literal and the complements of all preceding B-literals whose scope exceeds the number of A-literals between them and the terminal A-literal can be inferred.

The scope of each literal in the conclusion is 0 and the scope of each literal added to a chain in the extension and p-extension operations is 0. In the factorization and reduction operations (and also in ground factorization and reduction performed by the postprocessing operation), the scope of the leftmost involved literal is set to the maximum of its previous scope and the number of A-literals between it and the rightmost involved literal. In the p-reduction operation, the scope of a literal descended from an involved A-literal is set to the number of following A-literals in the derived chain. Each other literal has the same scope as its parent literal in the parent chain.

After each contraction operation, the scope of each literal is set to the minimum of its previous scope and the number of A-literals following it in the chain, i.e., no literal will be allowed to have a scope which exceeds the number of following A-literals.

Lemma formation creates assertions from solved goals. Removal of an A-literal by contraction does not mean that the goal it represents has been solved globally, but only that it has been solved in the environment of the chain of which it was a part. Joined in disjunction with the A-literal is the negation of each of the assumptions from the chain used in the solution of the A-literal. Thus, the resulting lemma states either the "solved" A-literal is true or one or more of the assumptions was false. The assumptions which could be used in the solution of the A-literal are the negation of an A-literal (reduction, p-reduction) or a B-literal (factorization). The scope mechanism keeps track of the assumptions made with respect to the solution of each goal.

Lemma formation, while it may generate useful assertions during the search for a proof, is not required for completeness.

Several heuristics are available to eliminate the generation of redundant lemmas such as: (1) the first lemma to be generated after an extension operation followed by zero or more factorization or reduction operations is always redundant and (2) if the chain has two or more terminal A-literals and the lemma associated with one subsumes the lemma associated with another,

the second need not be generated (specific cases of this condition can readily be checked by examining the scopes of the literals involved).

2.8.9. Subsumption. Redundant chains and assertions can be eliminated from future use by subsumption. One chain is subsumed by another and can thus be eliminated if an instance of the latter chain is an initial subsequence of the former (sequences of B-literals between A-literals may be freely reordered during the subsumption test). One assertion subsumes another if the chain corresponding to the former assertion subsumes the chain corresponding to the latter (the corresponding chain is formed by making a list of B-literals being the consequents and complemented antecedents of implications or disjuncts of disjunctions) and, in the case where the subsuming assertion is an implication, no disjunct or consequent of the subsumed assertion is matched to an antecedent of the subsuming assertion. The latter provision prevents the subsumption of an assertion by another implicative form of the same assertion both of which may be required for completeness. Stronger subsumption rules are possible (see [31]).

With relation to a search strategy, two additional classes of subsumption are recognized: (1) forward subsumption is the subsumption of a newly created chain or assertion by a previously available chain or assertion and (2) backward subsumption is the subsumption of a previously available chain or assertion by a newly generated chain or assertion. In general, completeness is

assured only if, when backward subsumption is used, it is first checked whether the subsuming chain is eliminable by forward subsumption.

Example 2.14. This and the following proofs illustrate the usage of most forms of the inference operations. Chains are represented as linear strings of literals with A-literals bracketed. A-literals represent "opened" goals, i.e., goals for which a solution is currently being attempted in the chain. B-literals represent "unopened" goals, i.e., subgoals for which an attempt for solution has not yet started. Each A-literal is a logical consequence of all the literals to its right; thus, the solution of each B-literal to the right of an A-literal solves the A-literal while also solving all the other A-literals to the right of the solved A-literal. Deductions are represented as a vertical sequences of chains, the ancestor of each derived chain being the chain above it. Each derived chain is annotated to describe its derivation from its ancestor. If a chain is the result of extension or p-extension by an assertion with more than one consequent or disjunct, an alphabetic index is used to designate which consequent or disjunct was used. Indices are "a", "b", "c", etc., reading from right to left. The unannotated chains at the top of each sequence of chains are the axioms. Here and elsewhere in this paper, the conclusion and assertions of a problem will frequently be referred to as theorem and axioms respectively.

This is a proof that $\neg(Pa \leftrightarrow Pb) \rightarrow a \neq b$.

1. $Pa \vee Pb$ — first axiom from $\neg(Pa \leftrightarrow Pb)$

2. $\neg Pa \vee \neg Pb$ — second axiom from $\neg(Pa \leftrightarrow Pb)$

3. $a \neq b$ — theorem to be established

4. $[a \neq b]\ \neg Pb\ \neg Pb$ — p-extend to 1b

   This operation initiates a proof by contradiction. Assuming $a=b$ (the complement of the literal $a \neq b$), the truth of $\neg Pb \wedge \neg Pb$ contradicts $Pa \vee Pb$.

5. $[a \neq b]\ \neg Pb$ — factor

   It is only necessary to prove $\neg Pb$ once.

6. $[a \neq b]\ [\neg Pb]\ Pa$ — extend by 2a

   By axiom 2, if Pa is true then $\neg Pb$ is true.

7. $[a \neq b]\ [\neg Pb]\ [Pa]\ Pb$ — p-reduce from A-literal

   Again assuming $a=b$ to derive a proof by contradiction, if Pb is true then Pa is true.

8. $\emptyset$ — reduce

   In chain 7, subject to the assumption that the conclusion $a \neq b$ is false and that $a=b$, there are the implications $Pb \rightarrow Pa$ and $Pa \rightarrow \neg Pb$. This leads to $Pb \rightarrow \neg Pb$, a contradiction if Pb is true. Therefore, $\neg Pb$ must be assumed to be true. From chain 5, $\neg Pb \rightarrow a \neq b$, so the theorem is proved.

The following four examples are adapted with some modification from [32]

Example 2.15.

1. $B \rightarrow C$
2. $A \wedge \neg D \rightarrow B$
3. $A \rightarrow C \vee \neg D$
4. A
5. C — theorem

6.	[C] B	extend by 1
7.	[C] [B] A -D	extend by 2
8.	[C] [B] A [-D] A -C	extend by 3a
9.	[C] [B] A [-D] A	reduce
10.	[C] [B] A	factor
11.	∅	extend by 4

Example 2.16.

1.	$b \leq a \wedge a \leq b \rightarrow a=b$	
2.	$b \geq 0$	
3.	$a \leq b$	
4.	$b \leq a$	
5.	$a \geq 0$	theorem
6.	[$a \geq 0$] $b \leq a$ $a \leq b$ $b \geq 0$	p-extend from 1
7.	[$a \geq 0$] $b \leq a$ $a \leq b$	extend by 2
8.	[$a \geq 0$] $b \leq a$	extend by 3
9.	∅	extend by 4

Example 2.17.

1.	$a>0 \rightarrow a \geq 0$	
2.	$\neg b \geq 0$	
3.	$a \geq 0$	
4.	$a \neq b$	theorem
5.	[$a \neq b$] $a>0$ $-b \geq 0$	p-extend to 1
6.	[$a \neq b$] $a>0$	extend by 2
7.	∅	extend by 3

Example 2.18.

1.	$a+b=2*c \rightarrow a \neq b \vee a=c$	
2.	$b \neq c$	
3.	$a+b=2*c$	
4.	$a \neq b$	theorem
5.	[$a \neq b$] $a+b=2*c$ $a \neq c$	extend by 1b
6.	[$a \neq b$] $a+b=2*c$ [$a \neq c$] $b \neq c$	p-reduce from A-literal
7.	[$a \neq b$] $a+b=2*c$	extend by 2
8.	∅	extend by 3

2.9. The Programmable Strategy Theorem Prover

The Programmable Strategy Theorem Prover (PSTP) is a program written in UCI LISP [8] for the DECsystem-10 computer implementing the linear MESON procedure.

The linear MESON procedure is a good inference system for an interactive, programmable strategy theorem prover since, it being an extension of the problem-reduction method, it is more human-oriented than alternative systems, and its input procedure nature and the relatively small number of operations that can be performed on any chain facilitates the design and use of the programmable search strategy capability.

In addition, the linear MESON procedure is a suitable choice in terms of performance since it appears to perform competitively with other inference procedures when used as the inference system in a fully automatic system. An implementation of the parent model elimination procedure at New York University [12] using a depth first search strategy performed competitively with a theorem prover employing the set of support refinement and unit preference search strategy. Further evidence of the competitiveness of linear MESON procedure based systems will be presented in the next section.

2.9.1. Chain properties. In the design of a theorem proving program, it is necessary to allow for the computation and retention of certain information about each chain (clause) generated during the search for a proof. An example is the necessity of retaining information on parentage of each chain so a proof can be traced when discovered. Another example is the computation of the length of a chain or the maximum level of function nesting in the chain if length or depth bounds are being employed. It would be wasteful to always compute and store such

information since it may not always be needed. Also, the retention of computed information about chains should be contingent on such variables as the computational effort required to compute the information, frequency of use of the information, and cost in memory of storing the information. (A more fundamental objection to always storing computed information about a chain is that the information might change with time. For example, the fact that a particular chain is the shortest chain in memory will probably be falsified in the future.) Another important consideration in the design of an information storage and retrieval mechanism for chains is the ability to define new data which can be optionally computed for any chain.

This last consideration is especially important in an interactive theorem proving program so that the user can cause to be computed whatever information about a chain will be useful to him. It is also an important consideration in the design of a theorem proving program which allows user specification of search strategies.

The property storage and retrieval mechanism for chains in PSTP was designed to possess the following characteristics. With the obvious exceptions of the number of a chain and its ancestry information, no information about a chain is computed unless and until this information is requested. Retaining the information is a user option. A new computable datum about a chain can be defined be merely defining the LISP function which computes the information.

The mechanism used is based on the concept of a chain property list. This is a list of dotted pairs; the first component of each dotted pair is a property name (the access name of a datum about the chain); the second component is the value of the property. The information storage and retrieval mechanism functions in the following way. If the value of the property named, for example, NLIT (this represents in PSTP the number of literals in the chain) is interrogated for a chain, that chain's property list is examined for a dotted pair with first component NLIT. If such a dotted pair is found, its second component is the desired information. If property NLIT does not appear in the chain property list, the LISP function NLIT is evaluated with the chain (including chain property list) as its single argument. The value the function NLIT returns is then the desired information. Further, if the LISP atom NLIT has non-NIL value, the property name NLIT and newly computed value will be added to the chain property list.

New properties are defined by the DP ("define property") function. The DP function takes as arguments a function name, lambda variable list, and expression (just like the UCI LISP DE, DF, and DM functions). It creates a LISP function which performs all the chain property list lookup and modification operations, and evaluates expression for the argument if the property value is not found on the chain property list. For example, NLIT is defined in PSTP by evaluating (DP NLIT (CHAIN) (LENGTH (CDDR CHAIN))) where LENGTH is the LISP function which computes the

length of a list and (CDDR CHAIN) is the location of the list of literals of chain CHAIN. (SETQ NLIT T) is then evaluated to order retention of values computed by the NLIT function.

Some of the property functions already defined in PSTP compute the number of A-literals in a chain (NALIT), the number of B-literals in a chain (NBLIT), the total number of literals in a chain (NLIT), the maximum function depth in a chain (DEPTH), the number of variables in a chain (NVAR), the number of LISP CONS operations required to construct a chain (SIZE) (this is a good size or complexity function), and the level (number of inference operations in the derivation) of a chain (NEXPAND).

2.9.2. Chain filters. This property storage and retrieval mechanism supports a higher level chain storage and retrieval mechanism. Filters provide a way of flexibly specifying which chains are to be operated upon and which derived chains are to be stored. Two types of filters are distinguished by usage: input filters and output filters. Input filters are employed by the user to specify which chains are to be operated upon. Only chains "selected by" an input chain filter will be processed. Output filters are used to specify which derived chains are to be retained. A chain must be "accepted by" an output filter to be stored. The general form for a filter is a unary LISP function name or lambda expression which returns a non-NIL value if the chain argument is selected or accepted, NIL otherwise. Several abbreviated forms are also available: (1) an integer selects or accepts a chain with that chain number, (2) a list of integers

selects or accepts chains with chain numbers in the list, (3) a three element list (called a triple) consisting of a binary function name and 2 integers or property names selects or accepts chains for which the value of the function applied to the integers and property values is non-NIL, and (4) a list of triples which selects or accepts chains for which each triple has non-NIL value.

This chain storage and retrieval mechanism is very flexible. The user can designate chains for processing directly by number or by the properties they possess and can arbitrarily specify the necessary conditions for a newly derived chain to be stored. This user specification of output filters is a far more general form of the usual specification of bounds in theorem proving programs.

2.9.3. Search strategy specification. One of the most important features of PSTP is its capability for specifying the search strategy to be used in searching for a proof. Several theorem proving programs (e.g., QA3.6 [36]) permit the user to specify a particular combination of refinements of resolution (restrictions on pairs of clauses to be used as input to the resolution operation (e.g., linear, merging, set of support, model refinements)), but the capability for ordering inference operations given a particular refinement of resolution is uncommon. PSTP is, of course, restricted to using the linear MESON procedure with variations restricted to different postprocessing operations, but it does have a general capability for specifying search strategies.

Before describing the search strategy specification capability of PSTP, it is instructive to consider the proof strategy employed in many other theorem proving systems in which search strategy is fixed with possibly a few parameters which the user can specify to tailor the proof search to a particular problem. Thus, the search strategy may be fundamentally depth first or perhaps breadth first with a parameter specifying the permitted amount of look-ahead using unit preference. Much of the control the user has over such systems is the specification of which chains to discard. However, even this decision is severely constrained. Usually, the user is only permitted to specify the values of a few parameters such as the maximum length or function depth of clauses to be retained.

We have seen that output filters generalize the capability of specifying retention of chains. Chain order functions provide the capability of specifying the order of expansion of the search space.

2.9.4. Chain order functions. Associated with each list of chains is the name of an order function. (The order function is actually a chain property function as described above.) Whenever a chain is stored in a chain list, it is inserted according to the numerical value of the corresponding order function applied to the chain. The chains with the smallest values of the order function are stored at the top of the chain list (in case of ties, the more recently stored chains will be on top of the chain list). This maintenance of chain lists in sorted order in

combination with the SEARCH and SEARCH2 commands provides a quite general capability for specifying search strategies.

2.9.5. Search commands. The SEARCH command is one of the fundamental functions for automatically expanding the search space. The normal mode of operation is for the SEARCH command to remove the top chain from a chain list, derive all possible immediate successor chains from this chain (this is known as expanding the chain), and store those successor chains selected by an output filter in the original chain list according to its order function. Thus by specifying an order function and using the SEARCH command, the user can specify in what order chains are to be expanded and thus partially control the search strategy. For example, if the default order function (which merely returns 0) is used, the search strategy is a depth first strategy. If the deduction level of a chain is used as order function, the search strategy is a pure breadth first strategy (level saturation). The SEARCH command can be viewed as an implementation of Nilsson's A^* algorithm [35] for graph searching as applied to theorem proving. Each derived chain is a node in the graph and the generation of all immediate successors to a chain by extension, factorization, etc., represents the expansion of a node in the A^* algorithm.

Although the SEARCH command is very effective in ordering the expansion of chains, the full expansion of a chain at each step often results in generating a large number of chains that will not be used because the value of the order function for these

chains exceeds the maximum order function value of any chain appearing in some proof. This presents two difficulties: (1) unused high order function valued chains fill up memory too quickly and (2) their generation requires extra, unnecessary work. The first of these problems could be solved by specifying an output filter that rejects chains with order function value exceeding a certain amount. However, this solution generates a bounded search strategy, i.e., a parameterized incomplete strategy which may fail to find a proof because the order function maximum is set too low. Moreover, the specification of a bounded search strategy fails to solve the problem of extra work required in the generation of rejected chains.

The solution adopted for PSTP includes a means for specification of ordering of individual inference operations rather than just chains. The form of the value of the order function was generalized to include sorted lists of operations with numerical values. For example, the order function value ((201 REDUCE) (302 EXTEND 6)) could represent the order function value for a chain with previously unperformed operations of reduction and extension by chain number 6. Chains with order function values of this form are inserted into chain lists according to the numerical value of the first specified operation (201 in the example).

The SEARCH2 command is designed to operate on chain lists with order functions of the new form. The SEARCH2 command, rather than deleting and expanding the top chain on the chain list,

deletes and performs only the first inference operation of the order function value of the top chain on the chain list. If any inference operations remain in the order function, the top chain is reinserted in the chain list according to the value of the next specified inference operation. Thus, the chain list is always a list of chains ordered according to the minimum value of the unperformed inference operations for that chain. The SEARCH2 command will generate successors of a single chain uninterruptedly only so long as none of the successors of the chain or any other chain on the chain list has an inference operation with lower numerical value than the next inference operation to be performed on the current chain.

The SEARCH2 command can also be viewed as an implementation of Nilsson's A^* algorithm for graph searching with each node being a chain and a single inference operation. Expansion of a node now merely consists of applying that inference operation to that chain.

Using the SEARCH2 command, for example, it is possible to use an order function which specifies a depth first search strategy that performs only one inference operation on each level (until a level bound is reached forcing a backup). A more realistic order function for use with the SEARCH2 command was used in the experiment described in 2.10. Note that the performance of the single specified inference operation may actually result in the generation of more than one successor chain as, for example, when there are two ways of extending a chain by a particular axiom.

2.9.6. Format functions. An additional mechanism for altering the strategy used by PSTP is accomplished by the use of format functions. Format functions can be used to reformat (edit) chains prior to their storage. For example, a format function can be used to reorder the last B-literals of a chain to accomplish the effect of Kowalski and Kuehner's literal selection function mechanism [25]. A format function is associated with each chain list. Each chain is reformatted according to the order function of the chain list into which it is being stored unless it is already in that format. A chain may be stored in two different formats in two different chain lists (formats are permitted to alter the sequence of literals which constitutes the chain itself, but not the chain property list). Default format functions are those which convert chains to x-standardized or y-standardized form by renaming variables.

2.9.7. Command summary. Following are brief descriptions of most of the PSTP commands. These PSTP commands can be divided into four classes: declarative commands (CHAINLIST, PARAMETERS, POSTPROCESSING, PROBLEM), informative commands (ANCESTRY, COUNT, DISPLAY), manipulative commands (COPY, DELETE, FOR, TRANSFER), and inference commands (EXPAND, SEARCH, SEARCH2). An abbreviated syntax for each command is also presented; linguistic variables are enclosed in angle brackets (e.g., "<sources>") and optional command phrases are enclosed in square brackets (e.g., "[DELETE]" and "[TO <destinations>]"). If any phrase of a command is absent, a default value will be used.

In the descriptions of the commands, the most important linguistic variables are <sources> and <destinations>. A source or destination represents, in general, a chain list and a chain filter. For a chain to be used by a command specifying <sources>, it must be a member of one of the specified chain lists and be selected by the corresponding chain filter. For a chain to be stored in the chain list of a destination by a command specifying <destinations>, it must be accepted by the corresponding chain filter. Note that in the syntax, <sources> and <destinations> are never optional (except as part of an optional phrase). This is because the empty specification for <sources> and <destinations> is legal and has a default value.

Any of the non-declarative commands can be interrupted at any time by typing any character. The command completes processing of the current chain and then enters a "break". In this state, the user can execute any PSTP command or LISP function and continue or abort the processing of the interrupted command.

2.9.8. ANCESTRY command. The (ANCESTRY [DELETE] <sources>) command prints the derivation of each chain designated by <sources>. If DELETE is specified, each designated chain is also deleted from its chain list.

2.9.9. CHAINLIST command. The (CHAINLIST [<declarations>]) command is used to declare chain lists that will be used and their format and order functions. If <declarations> is absent, the CHAINLIST command prints the list of previously declared chain lists and their format and order functions.

2.9.10. COPY command. The (COPY [DELETE] <sources> [TO <destinations>]) command copies each chain designated by <sources> to each of <destinations>. If DELETE is specified, each designated chain is also deleted from its chain list.

2.9.11. COUNT command. The (COUNT [DELETE] <sources>) command counts the number of chains designated by <sources>. If DELETE is specified, each designated chain is also deleted from its chain list.

2.9.12. DELETE command. The (DELETE <sources>) is the same as the (COUNT DELETE <sources>) command, i.e., the COUNT command with chain deletion specified.

2.9.13. DISPLAY command. The (DISPLAY [DELETE] <sources>) command prints each chain designated by <sources>. If DELETE is specified, each designated chain is also deleted from its chain list.

2.9.14. EXPAND command. The (EXPAND [EXTEND] [FACTOR] [REDUCE] [PEXTEND] [PREDUCE] [DELETE] <$sources_1$> [BY <$sources_2$>] [GIVING <$destinations_1$>] [AND <$destinations_2$>]) is the principal inference command for interactive use. It will perform the designated inference operations on each of the chains designated by <$sources_1$> using each chain designated by <$sources_2$> as second argument to binary inference operations (extension and p-extension). Derived chains will be stored in <$destinations_1$> and lemmas will be stored in <$destinations_2$> (chain filters

permitting). If DELETE is specified, each designated chain in <sources$_1$> is also deleted from its chain list. The EXPAND command is restricted to performing inference operations on chains existing at the time of its invocation, i.e., it will not perform any inference operations on chains it has just derived. The command terminates when (1) the empty chain is generated, i.e., a proof has been found, (2) all the specified operations have been performed, or (3) the user suspends processing of the command by typing any character. If no inference operations are designated, all inference operations will be used. If at least one inference operation is designated, the word EXPAND may be omitted.

2.9.15. FOR command. The (FOR [DELETE] <sources> DO <function>) command applies the unary LISP function <function> to each chain designated by <sources>. If DELETE is specified, each designated chain is also deleted from its chain list.

2.9.16. PARAMETERS command. The (PARAMETERS [<index>]) command is used to declare the values of several global parameters. If an <index> (an arbitrary LISP atom) is specified, it designates for use the predefined set of parameter values associated with <index>. If an <index> is not specified or <index> has no previously defined meaning, the PARAMETERS command asks a series of questions requiring the user to define the value for each parameter. Parameters set by the PARAMETERS command include: whether newly generated chains are to be printed, the

format in which chains are to be printed, whether lemmas are to be generated, and whether subsumption is to be performed.

2.9.17. POSTPROCESSING. The (POSTPROCESSING [<index>]) command is used to declare what postprocessing operation is to be employed. If an <index> (an arbitrary LISP atom) is specified, this <index> designates the postprocessing operation that will be used. Allowed <index>s include WEAK-SAVE, WEAK-DELETE, STRONG-SAVE, and STRONG-DELETE, designating the postprocessing operations described previously. If an <index> is not specified or <index> has no previously defined meaning, the POSTPROCESSING command asks a series of questions requiring the user to designate which action among a list of alternative actions is to be taken for a given condition. For example, the POSTPROCESSING command may ask whether, in the case of an A-literal followed by an identical B-literal, the B-literal should be saved, deleted, or deleted with the reduction operation recorded in the ancestry of the chain.

2.9.18. PROBLEM command. The (PROBLEM [<declarations>]) command sets up a problem for the theorem prover. It first makes the chainlist declarations specified by <declarations> (if fewer than two chain list declarations are specified, up to two default declarations will be made) and then asks the user to type in the theorem and each axiom. The theorem is stored in the first declared chain list; its negation and the axioms are stored in the second declared chain list. The input format for the theorem

and axioms is the same as was used in the description of the linear MESON procedure except that prefix form for predicate and function symbols is required, and (due to character set limitations) $\wedge$ and $\vee$ are omitted and --> is substituted for $\rightarrow$. Thus, Pab $\wedge$ Pba $\rightarrow$ a=b $\vee$ Qabx is typed in as Pab Pba --> =ab Qabx. The theorem and axioms are then encoded into internal list form. The PROBLEM command permits the user to save the encoded axioms so that they will not need to be retyped in future proofs of the same problem.

2.9.19. SEARCH command. The (SEARCH [EXTEND] [FACTOR] [REDUCE] [PEXTEND] [PREDUCE] <sources$_1$> [BY <sources$_2$>] [GIVING <destinations$_1$>] [AND <destinations$_2$>]) repeatedly deletes the first chain from <sources$_1$> (a chain with lowest order function value) and performs on it each designated inference operation with the chains designated by <sources$_2$> as second argument to binary inference operations (extension and p-extension). Derived chains will be stored in <destinations$_1$> and lemmas will be stored in <destinations$_2$> (chain filters permitting). So that newly generated chains can be used as input to inference operations by the SEARCH command, <sources$_1$> and <destinations$_1$> will ordinarily specify the same chain lists. The command terminates when (1) the empty chain is generated, i.e., a proof has been found, (2) <sources$_1$> is empty meaning no more operations can be performed and no proof could be found within the constraints of the specified operations, initial chains, and chain filters, or (3) the user suspends processing of the command

by typing any character. If no inference operations are designated, all inference operations will be used.

2.9.20. SEARCH2 command. The (SEARCH2 [EXTEND] [FACTOR] [REDUCE] [PEXTEND] [PREDUCE] <sources$_1$> [BY <sources$_2$>] [GIVING <destinations$_1$>] [AND <destinations$_2$>]) repeatedly deletes the first chain from <sources$_1$> (a chain with lowest order function value) and performs on it the first designated inference operation in the order function value. This inference operation is then deleted from the order function value and, if any inference operations remain in the order function value, the chain is reinserted in <sources$_1$> (now with the numerical value associated with the next inference operation as the numerical value of the chain for insertion into the sorted chain list). Derived chains will be stored in <destinations$_1$> and lemmas will be stored in <destinations$_2$> (chain filters permitting). So that newly generated chains can be used as input to inference operations by the SEARCH2 command, <sources$_1$> and <destinations$_1$> will ordinarily specify the same chain lists. The order function, using variables of the SEARCH2 function, will construct a list of inference operations with (in the case of binary inference operations) second arguments as specified in <sources$_2$> for derived chains as they are stored. The command terminates when (1) the empty chain is generated, i.e., a proof has been found, (2) <sources$_1$> is empty meaning no more operations can be performed and no proof could be found within the constraints of the specified operations, initial chains, and chain filters, or

(3) the user suspends processing of the command by typing any character. If no inference operations are designated, all inference operations will be used.

2.9.21. TRANSFER command. The (TRANSFER <sources> [TO <destinations>]) command is the same as the (COPY DELETE <sources> [TO <destinations>]) command, i.e., the COPY command with chain deletion specified.

2.10. PSTP performance study

In order to give some idea of the performance of PSTP with some simple search strategies and to make some points about relative merits of some of these strategies, the results of PSTP runs on 9 examples using 4 strategies are presented here. Results are compared to results for two other theorem proving programs tested on the same examples.

2.10.1. The problems. The problems are taken from a comparative study of theorem proving strategies used by QA3.6 by Reboh et al [36] (additional information on sources, theory, and previous uses of these examples are in [36]); the same examples were also run for an SL-resolution theorem prover (here called SLRTP) by Aubin [1,2]. The examples are axiomatized just as for QA3.6 with an occasional substitution of a disjunction for an implication and, in the cases of unsatisfiable sets of axioms, the use of the negation of one of the axioms as the theorem. Appendix A contains a list of the problems, results, and proofs.

2.10.2. Inference operations used. All the problems were run with extension as the only rule of inference except the NUM1 example for which reduction was also necessary. The WEAK-DELETE postprocessing operation was used for all the examples. Its use, of course, permits ground factorization and reduction. In some examples (BURSTALL, SHORTBURST, GROUP1, GROUP2), it is readily apparent from the structure of the problem that no reduction is possible (since every chain derived from the theorem has only positive literals eliminating any possibility of matching an A-literal with a complementary B-literal). The ANCES1 example is propositional and thus the ground reduction in the WEAK-DELETE postprocessing operation is sufficient. In the remaining three problems (HAS-PARTS1, HAS-PARTS2, PRIM) for which reduction was not employed (although ground reduction was used in each), the use of the reduction operation resulted in the generation of no additional chains. Lemmas were not generated for any of the examples.

2.10.3. Search strategies used. The strategies used are characterized by 4 parameters: length multiplier, level multiplier, length maximum, and level maximum. The length of a chain is defined to be its number of B-literals. This is consistent with the notion of the length of a clause in resolution theorem proving being its number of literals since in a chain A-literals record ancestry information and would not be present in the corresponding clause form. The level of a chain is defined to be the number of inference operations employed in

deriving it from the alleged theorem excluding those operations (ground factorization and reduction) automatically performed by the postprocessing operation.

The SEARCH2 search command was employed with projected inference operations ordered according to the minimum values of a weighted sum of the expected length and level of the result. The expected length of a chain derived by extension is the length of its parent chain being extended plus the length of the axiom minus 2. The expected length of a chain derived by factorization or reduction is the length of its parent chain minus 1. The actual length may be less (but never more) due to removal of B-literals by the accepting transformation. The expected and actual level of a chain is the level of its parent plus 1. Only inference operations whose results have expected lengths and levels not exceeding the length or level maxima will be attempted (this way of implementing length and level maxima was also used by QA3.6 and SLRTP).

Two sets of length and level multipliers were tried. The first has a length multiplier of 101 and a level multiplier of 100 and is called the 101/100 strategy. In the 101/100 strategy, the projected inference operation with highest merit is one with the smallest value of (100 times) the sum of expected length and level of the result. Ties are resolved in favor of lesser expected length (a 100/101 strategy would resolve ties in favor of lesser expected level). (It is assumed here that the expected length of a chain will never exceed 100.) The most important

thing to note about the 101/100 strategy is that it is essentially the same as Kowalski and Kuehner's upper diagonal search strategy [25]. It is an admissable strategy [35] except for cases where the postprocessing operation removes B-literals by ground factorization or reduction. First proofs discovered by admissable strategies are guaranteed to be minimum level proofs.

The second strategy has a length multiplier of 501 and a level multiplier of 100 and is called the 501/100 strategy. In the 501/100 strategy, the projected inference operation with highest merit is one with the smallest value of (100 times) the sum of the expected level and 5 times the expected length of the result. Ties are again resolved in favor of lesser expected length. By multiplying length by 5 times as much as level, a strong length preference strategy is produced. The 501/100 strategy is, of course, inadmissable since it is clearly not always the case that it requires at least 5 inference operations to remove a single literal. (For a strategy to be admissable, the estimated additional cost to solution must always be less than or equal to the actual additional cost to solution.)

The 101/100 and 501/100 strategies were each tried with (bounded) and without (unbounded) length and level maxima. The length and level maxima used were those used by QA3.6 wherever possible.

2.10.4. Statistics. The performance of strategies will be primarily characterized by the "chains generated" statistic. Here, this information is represented by a 4-tuple: the first component is the number of chains retained; the second component is the number of acceptable chains generated; the third component is the total number of chains generated; the fourth component is the number of attempted inference operations. The number of retained chains is the number of acceptable chains minus the number of chains eliminated by subsumption, function depth tests, etc. No such processes were used to eliminate chains in this experiment, so the number of retained chains is always equal to the number of acceptable chains. The total number of chains generated is the number of acceptable chains plus the number of non-acceptable chains generated. These statistics and the time figures referred to below are automatically accumulated by PSTP and printed out when a proof is found.

Nearly comparable statistics are presented where available for QA3.6 and SLRTP (except QA3.6 statistics refer to clauses rather than chains). Data for the number of acceptable chains generated is summarized in Table 2.1 as well as being available in Appendix A.

Best and mean performance figures are presented for QA3.6 on each example. For QA3.6, the number of retained clauses is the number of retained clauses after subsumption and function depth tests; the number of acceptable clauses is computed as the number of successful resolutions and factorings; the number of attempted

Problem	Acceptable Chains Generated: unbounded 101/100	unbounded 501/100	bounded 101/100	bounded 501/100	best QA3.6	mean QA3.6	SLRTP
1. BURSTALL	191	74	45	45	42	118	48
2. SHORTBURST	18	16	16	16	12	21	16
3. HAS-PARTS1							
implicative	7	7	7	7			
disjunctive	12	12	12	12	10	24	12
4. HAS-PARTS2							
implicative	11	11	11	11			
disjunctive	50	38	38	38	14	51	20
5. PRIM							
implicative	812	57	70	54			
disjunctive	165	101	130	101	19	97	134
6. ANCES1	23	13	13	13	12	13	14
7. NUM1	10	10	10	10	10	11	9
8. GROUP1	14	14	14	14	7	7	12
9. GROUP2	576	119	225	225	74	82	?

Table 2.1

inference operations is computed as the number of attempted resolutions and factorings. The proportion of tested QA3.6 strategies which discovered a proof is given on the same line as the mean performance of QA3.6 strategies; unsuccessful strategies were excluded in computing the means.

Performance figures are presented for SLRTP where the set of support for the refutation was the negation of the theorem in the PSTP proof on each example. Due to the similarity of operations and terminology between the linear MESON procedure and the

inference system for SLRTP, SL-resolution [25], we will here present a brief description of SL-resolution.

SL-resolution, a refutation procedure, can be viewed as a variant of model elimination without equality with the following features. (1) The capability for reordering B-literals at the end of a chain is formalized in the form of a literal selection function which designates the literal to be extended on in succeeding extension operations. (2) Factorization is a required operation for completeness in SL-resolution since the equivalent of the STRONG-SAVE postprocessing operation is employed. The model elimination factorization and reduction operations are combined into the SL-resolution reduction operation. (3) SL-resolution requires a fully factored input set of clauses, i.e., every non-tautologous factor of an input axiom must also be input (or, as in SLRTP, derived). A benefit of this is that SL-resolution reduction operations need never be performed with the leftmost involved literal being or following the last A-literal of the chain. (4) Any B-literal following the last A-literal of the chain is a candidate for removal by the reduction operation, not just the rightmost as in model elimination. (5) Upper diagonal search is the prescribed search strategy for SL-resolution.

For SLRTP, the number of retained chains is the number of retained chains after function depth tests and subsumption (subsumption is only used in eliminating redundant axiom chains or their factors during the process of generating a fully

factored input set of axioms); the number of acceptable chains is computed as the number of successful extensions, reductions, and factorings (used only for generating a fully factored input set of axioms); the number of attempted inference operations is computed as the number of attempted extensions, reductions, and factorings. The GROUP1 and GROUP2 example statistics are taken from [2].

The "search time" statistic represents the time spent in searching for a proof by a compiled version of PSTP; it excludes time spent in inputting the problem, outputting of final statistics and proof, and garbage collection, although it does include time required for some trace output during the search. Search time is the only widely variable component of total time to solution with problem input and statistics and proof output time relatively constant and small. Although PSTP is conservative of storage (performing LISP CONS operations only when necessary when instantiating chains) and therefore ordinarily requires few garbage collections, garbage collection time is excluded because (1) time consuming garbage collections occurring at random times in the search for a proof tend to randomize the time statistics especially for short searches (this problem could be overcome by always starting a search for a proof immediately after a garbage collection) and (2) frequency of garbage collection is dependent on the amount of storage available (with infinite storage, there need not be any garbage collections). Nearly all the proofs presented here were found

with about 25000 words available for storing chains and most were found with no garbage collections.

Time statistics should not be used for comparison among strategies used by different theorem provers without considerable caution and more information than is usually available. Such statistics are of course influenced by the machine and operating system used, language and coding of the theorem prover, whether compiled (SLRTP, PSTP) or interpreted (QA3.6), special conditions applying to the operation of the theorem prover (e.g., tracing), and some randomness in the times themselves (such randomness, attributable to variable load on the time sharing system, is visible in some anomalies in the statistics presented here).

2.10.5. Results. Four primary observations can be made from the results of the experiment presented here: (1) PSTP performs competitively with QA3.6 and SLRTP, (2) the 501/100 strategy performs better than the 101/100 strategy (for these examples), (3) the 501/100 strategy is relatively insensitive to length and level bounds, the 101/100 strategy is much more sensitive, and (4) elimination of some implicative forms of the axioms can result in improved performance.

The basis for comparison of the results of PSTP and QA3.6 is the number of acceptable chains generated (equals the number of chains retained) for PSTP versus the number of acceptable clauses generated (equals the number of successful resolutions and factorings) for QA3.6. This is a fairer comparison than one using the number of retained clauses for QA3.6 since QA3.6

eliminated clauses by function depth maxima and subsumption. Even this comparison is still somewhat unfair to PSTP since if function depth tests and subsumption had not been used in QA3.6, the number of generated clauses would presumably have been larger since eliminated clauses could now act as parent clauses in additional inferences.

Using this basis for comparison, the unbounded 501/100 strategy (the strategy we prefer for reasons given below) performed better than the average of QA3.6 strategies which found a proof in all the examples except PRIM, GROUP1, and GROUP2.

In the PRIM example, the unbounded 501/100 strategy performed only slightly worse than the average of QA3.6 strategies.

In the GROUP1 example, the absolute difference in performance is small even if the number of chains generated by PSTP is double the number of clauses generated by QA3.6. In view of the fact, for example, that by reversing the order of presentation of the axioms to PSTP can cause the performance of PSTP to exceed that of QA3.6, we tend to regard this difference as being relatively insignificant.

The difference in the case of GROUP2 is much more serious and has a rather different explanation. Where the formulation of the GROUP2 example has several unit axioms and two 4-literal associative axioms, the use of a length maximum value of 3 can be seen to be extremely restrictive. In resolution terms, this length maximum requires that only units be resolved against the associative axioms, and in the case of GROUP2 if the negation of

the theorem is used as the set of support, the length maximum automatically restricts any tested strategy to a further refinement of unit resolution in which only the negation of the theorem can be directly resolved against the associative axioms. PSTP was tested with a variant of the GROUP2 example in which axiom 3 was reordered so that a proof meeting the length maximum value of 3 restriction existed. On this example, the unbounded 101/100 strategy generated 435 chains; all the other strategies generated 29. We therefore feel that the better performance of QA3.6 on this problem was more attributable to the restrictive length maximum than to an intrinsic inferiority of PSTP.

The comments about the restrictive length bound used by QA3.6 in the GROUP2 example can be extended to several other examples. The BURSTALL, SHORTBURST, HAS-PARTS1, HAS-PARTS2, PRIM, ANCES1, GROUP1, and GROUP2 examples all had very restrictive length maxima, in every case set at or below the minimum value required for PSTP to discover a proof. Level maxima were often similarly restrictive although we perceive this to be much less important in reducing the size of the search in the non-depth first search strategies tested. We feel that use of such restrictive length and level (especially length) bounds invalidates the results of [36] to a degree, since their use imposes severe limitations on the structure of the search space. In this restricted search space, tests of different strategies may fail to discriminate between strategies, or unfairly discriminate between them.

Given the similarity of SL-resolution and the linear MESON

procedure, one would anticipate substantial similarity in the results for PSTP using the bounded 101/100 strategy (upper diagonal search) and SLRTP. For 6 of the examples (BURSTALL, SHORTBURST, HAS-PARTS1, PRIM, ANCES1, and NUM1), the results agree closely. Differences emerge for the remaining 3 examples. We don't know why PSTP did so much worse than SLRTP on the HAS-PARTS2 example. In the GROUP1 example, use of a fully factored input set of clauses was clearly beneficial to SLRTP since the very short proof could be shortened further by using a factored form of one of the associative axioms. PSTP with factorization could not match the SLRTP results, since extension by the associative axiom followed by factorization counted as 2 inference operations in computing the level of the resulting chain (whose value is used to compute the order function value) whereas extension by the factored associative axiom by SLRTP counts as only 1 inference operation. (PSTP could be made to equal SLRTP's performance on this example by inputting a fully factored set of axioms, a perfectly legal operation, although unnecessary for completeness.) In the case of the GROUP2 example, for which SLRTP failed to find a solution, both the bounded and unbounded 101/100 strategies in PSTP showed relative difficulty in discovering a solution. In SLRTP, this difficulty was exacerbated by the very feature which aided the quick solution of the GROUP1 example: mandatory factorization. PSTP with factorization and the STRONG-DELETE postprocessing operation (resulting in an inference system very similar to SL-resolution)

failed to find a proof with the unbounded 101/100 strategy after 1507 chains were generated, discovered a proof while generating 410 chains with the unbounded 501/100 strategy, and discovered a proof while generating 458 chains with each of the bounded strategies. The proofs discovered were the same as those discovered without factorization.

We believe the detrimental effects of factorization as demonstrated in the GROUP2 example results are more typical than the beneficial effects illustrated in the GROUP1 example. In our experience, even in cases where factorization does shorten a proof (as it did not in the GROUP2 example), the proliferation of highly instantiated chains caused by the use of factorization still often outweighs the benefits. (These negative comments clearly refer only to general factorization where literals must be unified; factorization in the ground case is clearly beneficial and is included in the postprocessing operations we used here.) Should future experience prove this judgment about factorization wrong, the linear MESON procedure still permits factorization as a legal though optional operation.

Another point can be made here concerning SLRTP's efforts to discover a solution to the GROUP2 example. SLRTP uses a literal selection function to designate which literal of each derived chain is to be used in future extension operations. The only literal selection function tested was the function which always selects a literal which has the fewest matching literals among the axioms. This has the obviously desirable characteristic of

reducing the branching rate of the search tree since the selected literal has the fewest matches among the axioms and, further, removal of the selected literal after some inference operations will usually instantiate the remaining literals and reduce the number of literals among the axioms matching them. However, this literal selection function is, in the case of problems with structure similar to the GROUP2 example, inconsistent with the use of length maxima. In GROUP2, for example, the literal selection function will show a preference for literals capable of being extended upon only by the associative axioms (since any positive literal matches the consequent of the associative axioms, any literal matching a unit axiom also matches the associative axioms). Thus, the effect of the use of this literal selection function is to increase the length of chains appearing in a deduction possibly requiring the increase of the length maximum used.

One final point remains about the comparison of results between PSTP and SLRTP. This concerns the very small number of attempted inference operations by SLRTP. This is due to the use of a literal classification tree which automatically selects out likely matches for literals to be extended upon from among the literals in the axioms. The extension operation is only attempted for axioms containing literals selected by the literal classification tree. This probably represents a fairly small (though real) saving in computational effort since one must count the cost of creating and accessing the literal classification

tree and the cost saved is that of attempting unifications destined to fail, usually a fairly quick operation. The real benefit of use of the literal classification tree is the elimination of the multiple attempts at unifying literals that would ordinarily result from use of the literal selection function requiring discovery of the number of matches for a literal among literals in the axioms.

In comparing the four strategies tested by PSTP among themselves, one first discovers that the 501/100 strategy invariably performed as well as or better than the 101/100 strategy for the same choice of length and level maxima. This is especially true of the results for the BURSTALL, GROUP2, and PRIM examples in the absence of length and level maxima. A further demonstration of the superiority of the 501/100 strategy is its relative insensitivity to length and level bounds. Only in the BURSTALL example did the bounded 501/100 strategy perform significantly better than the unbounded strategy. Also, in the PRIM and GROUP2 examples, the addition of length and level bounds actually degraded the performance of the 501/100 strategy since the bounds excluded proofs discovered by the unbounded strategy. In contrast, performance of the 101/100 strategy was often improved by the addition of length and level bounds, but (as stated above) never improving upon the performance of the 501/100 strategy. The demonstrated insensitivity of the 501/100 strategy to the addition of length and level bounds seems especially significant in view of the often extreme restrictiveness of the bounds tested.

Due to its generally good performance and lack of improvement with the addition of bounds, we regard the unbounded 501/100 strategy as the best among those tested.

We feel generally that, provided it performs adequately, a complete (e.g., length preference) strategy like the unbounded 501/100 strategy is to be preferred to an incomplete (e.g., length bounded) strategy like the bounded 101/100 strategy, even if the latter, with appropriate choice of bounds, can often match the performance of the former.

Finally, we merely note that judicious elimination of various implicative forms of the axioms can result in significantly improved performance as demonstrated in the results for the HAS-PARTS1, HAS-PARTS2, and PRIM examples. Of course, this elimination of implicative forms of the axioms destroys the completeness property of the linear MESON procedure. However, this controlled incompleteness may be desirable in cases where significant improvement in performance results. Completeness could be preserved and nearly the same effect gained by presenting PSTP with all the implicative forms of the axioms, but (via the order function definition) giving PSTP a strong preference for using one instead of another.

One feature of the linear MESON procedure not previously discussed is the length of its proofs. It is characteristic of linear theorem proving strategies that they require longer proofs than some other strategies. This and past studies [1,12] indicate that linear strategies can overcome this increased proof

length and perform competitively with other procedures. The linear resolution strategy tested in QA3.6 was less successful since the special chain rejection criteria of variants of the model elimination procedure were not used.

While the length of a proof is one measure of its complexity, we feel that the increased length of linear MESON procedure proofs is not a great disadvantage in terms of readability. The problem-reduction method oriented form of such proofs often makes them more comprehensible than ordinary resolution proofs relying on converging lines of deduction resulting in a refutation.

2.11. π theorem proving procedure

The π theorem proving procedure is a new variant of the MESON procedures which is strongly related to the definition we gave for the problem reduction method.

Where the problem reduction method has a single inference operation (the problem reduction operation), the π procedure has a single inference operation (the π reduction operation). In the problem reduction method, generated subgoals are solved using only the original assertions of the problem. In the π procedure, generated subgoals are solved using the original assertions of the problem plus the negation of the generating goal. This modification is all that is required to extend the problem reduction method to completeness for the propositional calculus and with appropriate qualification to the first order predicate calculus.

The $*$ procedure makes no assumptions about the data structure or format of partial deductions, but only defines the single inference operation of using implications to generate subgoals and defining the context in which these subgoals must be proved. The MESON procedures can be regarded as forms of the $*$ procedure which make assumptions about the data structure: trees for the MESON procedure, chains for the linear MESON procedure. They also incidentally differentiate between inference operations using implications of the original problem and those using higher goals. In the $*$ procedure, the two types of assertions (assertions of the original problem and negations of higher goals) are treated uniformly.

We will treat in detail only the propositional calculus form of the $*$ procedure with the understanding that it can readily be lifted to the first order predicate calculus by conventional techniques such as those used for the resolution and model elimination theorem proving procedures.

Following is a definition of the $*$ procedure in terms of its single inference operation, the $*$ reduction operation.

Definition 2.30. $B_1,\ldots,B_p \vdash_* C$ if

B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C$ $(m \geq 0)$

and $B_1,\ldots,B_p,\sim C \vdash_* A_1$

:

and $B_1,\ldots,B_p,\sim C \vdash_* A_i$

:

and $B_1,\ldots,B_p,\sim C \vdash_* A_m$ for some k $(1 \leq k \leq p)$.

The π procedure is so named because (1) in physics the π meson is a type of meson thus suggesting the relationship to the MESON procedure, (2) π spelled out in Latin letters in π suggesting Premises and implications, the two types of assertions used in the MESON procedure, and (3) the Greek letter π corresponds to the Latin letter p so that the π reduction operation suggests the problem reduction operation.

Theorem 2.4. The π procedure is sound, i.e., if $B_1,\ldots,B_p \vdash_\pi C$ then $B_1,\ldots,B_p \vdash C$ where $\vdash_\pi$ denotes derivability by the π procedure.

Proof. We will perform induction on the number of applications n of the single π procedure inference rule used in a π procedure proof.

n=1. Assume $B_1,\ldots,B_p \vdash_\pi C$ with only one application of the inference rule used in the proof. Since only one application of the inference rule is required, B_k must be C itself. Otherwise, additional applications would be required to prove A_1, ..., and A_m. Clearly $B_1,\ldots,B_{k-1},C,B_{k+1},\ldots,B_p \vdash C$. Therefore, the π procedure is sound for this case.

n>1. Assume the π procedure is sound for all proofs with fewer than n applications of the inference rule. Prove it is sound for n applications. Assume $B_1,\ldots,B_p \vdash_\pi C$ using n applications of the inference rule.

The proof of C employs the subproofs $B_1,\ldots,B_p,\sim C \vdash_\pi A_1$, ..., $B_1,\ldots,B_p,\sim C \vdash_\pi A_i$, ..., and $B_1,\ldots,B_p,\sim C \vdash_\pi A_m$ where B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C$ for some k ($1 \leq k \leq p$). By the induction hypothesis, since

each of these subproofs has fewer than n applications of the inference rule, $B_1,\ldots,B_p,\sim C\vdash A_1$, ..., $B_1,\ldots,B_p,\sim C\vdash A_i$, ..., and $B_1,\ldots,B_p,\sim C\vdash A_m$. Thus, $B_1,\ldots,B_p,\sim C\vdash A_1 \wedge \ldots \wedge A_m$ and, by the deduction theorem, $B_1,\ldots,B_p\vdash \sim C \rightarrow A_1 \wedge \ldots \wedge A_m$. Taking the contrapositive of the theorem, we get $B_1,\ldots,B_p\vdash \neg(A_1 \wedge \ldots \wedge A_m) \rightarrow C$. But, since B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C$ we also have $B_1,\ldots,B_p\vdash A_1 \wedge \ldots \wedge A_m \rightarrow C$. Therefore, $B_1,\ldots,B_p\vdash(A_1 \wedge \ldots \wedge A_m) \vee \neg(A_1 \wedge \ldots \wedge A_m) \rightarrow C$ and, by simplification, $B_1,\ldots,B_p\vdash C$. Therefore, the π procedure is sound for this case. □

In order to prove the completeness of the π procedure, we will first prove the following lemma. The lemma relates to a procedure related to the π procedure which operates on clauses rather than on implications. Thus, at each step of the procedure, rather than matching a goal to the consequent of an implication and forming subgoals of the antecedents of the implication, the procedure of the lemma matches any literal of a clause and forms subgoals of the negated remaining literals of the clause. By the definitions of the $\vee$, $\wedge$, and $\rightarrow$ connectives, it can be readily seen that the two processes are equivalent.

The reason for proving completeness results in terms of clauses rather than more general assertions is that if implications are used, two different but logically equivalent (meaning they are represented by the same clause) may be necessary in the proof. This need for different (but logically indistinguishable) assertions would complicate the completeness proofs.

The assertion $A_1 \wedge \ldots \wedge A_m \rightarrow C_1 \vee \ldots \vee C_n$ can be converted to the clause $\sim A_1 \vee \ldots \vee \sim A_m \vee C_1 \vee \ldots \vee C_n$.

The lemma concerns fully unit subsumed sets of clauses. A fully unit subsumed set of clauses is a set of clauses in which there is no clause which is subsumed by a unit clause in the set (except, of course, for the unit clause itself). This is not a requirement of the procedure, but aids in the proof of completeness. The use of fully unit subsumed sets of clauses does have implications for some features of the procedure, i.e., the elimination of goals which are complementary to higher goals and the elimination of goals which are instances of axioms, which will be noted later.

Lemma 2.1. Assume $\{B_1,\ldots,B_p\}$ is a fully unit subsumed set of clauses not including the clause $\sim C$ and $\{B_{k_1},\ldots,B_{k_{p'}},\sim C\}$ is a minimally inconsistent set for some subset $\{B_{k_1},\ldots,B_{k_{p'}}\}$ of $\{B_1,\ldots,B_p\}$. Then $B_1,\ldots,B_p \vdash_* C$.

Proof. Induction will be performed on the number of non-unit clauses k among the clauses $B_1,\ldots,B_p$.

Case 1. k=0. If k=0, then $C \in \{B_1,\ldots,B_p\}$ and $B_1,\ldots,B_p \vdash_* C$ is immediate and the lemma is true for this case.

Case 2. k>0. Assume the lemma is true for all sets of clauses with number of non-unit clauses less than k. Prove it is true for all sets of clauses with number of non-unit clauses equal to k.

Assume $\{B_1,\ldots,B_p\}$ includes k non-unit clauses.

Case 2a. $C \in \{B_1,\ldots,B_p\}$. $B_1,\ldots,B_p \vdash_* C$ is immediate and the lemma is true for this case.

Case 2b. $\sim C \in \{B_1,\ldots,B_p\}$. This is contrary to the assumption of the lemma and thus cannot occur.

Case 2c. Neither C nor $\sim C \in \{B_1,\ldots,B_p\}$. Since $\sim C$ is in a minimally inconsistent set $\{B_{k_1},\ldots,B_{k_p},\sim C\}$ by hypothesis, some B_{k_j} must be of the form $A_1 \vee \ldots \vee A_{i-1} \vee C \vee A_{i+1} \vee \ldots \vee A_m$.

Each $\{B_{k_1},\ldots,B_{k_{j-1}},A_{i'},B_{k_{j+1}},\ldots,B_{k_p},\sim C\}$ is inconsistent $(1 \leq i' \neq i \leq m)$. Further, each $A_{i'}$ is in a minimally inconsistent subset of this set.

Let B be the set $\{B_1,\ldots,B_p,\sim C\}$ with all clauses containing $\sim C$ except the unit clause $\sim C$ itself removed. B is fully unit subsumed by virtue of the removal of all clauses containing $\sim C$ except $\sim C$ itself and the fact that $\{B_1,\ldots,B_p\}$ is fully unit subsumed. The minimally inconsistent set $\{B_{k_1},\ldots,B_{k_p},\sim C\}$ contains none of the clauses subsumed by $\sim C$ by minimality. Thus, the subset of $\{B_{k_1},\ldots,B_{k_{j-1}},B_{k_{j+1}},\ldots,B_{k_p},\}$ for each $A_{i'}$ which with $A_{i'}$ forms a minimally inconsistent set is also a subset of B. No $A_{i'} \in B$ since none was in $\{B_1,\ldots,B_p\}$ since $\{B_1,\ldots,B_p\}$ is assumed to be fully unit subsumed and $A_{i'}$ subsumes B_{k_j}. Thus B meets all the assumptions of the lemma and has less than k non-unit clauses. By the induction assumption, $B \vdash_\pi \sim A_{i'}$ for each $A_{i'}$. The proof of each $\sim A_{i'}$ constitutes a proof of C and the lemma is true for this case. □

Theorem 2.5. The π procedure is complete, i.e., if $B_1,\ldots,B_p$ are consistent and $B_1,\ldots,B_p \vdash C$ then $B_1,\ldots,B_p \vdash_\pi C$

Proof. Let B be the subset of $B_1,\ldots,B_p$ such that $B \cup \{\sim C\}$ is a minimally inconsistent set. There must be such a set since

$B_1,\ldots,B_p$ are consistent and imply C. By the lemma, $B \vdash_\pi C$. $B_1,\ldots,B_p \vdash_\pi C$ since a proof of C from B is also a proof of C from $\{B_1,\ldots,B_p\}$. □

The π procedure was expressed in terms of a single inference rule (π reduction) with no limitations on its application. The soundness theorem supports the validity of the use of the π reduction rule without restriction. However, the completeness theorem suggests several restrictions which can be used to substantially improve the efficiency of the π procedure. These restrictions are enumerated below.

In a π procedure deduction, define an opened goal to be a goal to which π reduction has been applied resulting in at least one subgoal; define an unopened goal to be a goal without subgoals.

1. No goal may have as a subgoal (immediate or otherwise) an identical goal (opened or unopened). This is equivalent to the model elimination admissability rules that no chain with an A-literal followed by an identical B-literal needs to be retained.

The justification for this in the π procedure is that (examining the proof of the completeness theorem) fully unit subsumed sets of clauses are used. Thus, if A is a higher goal, the completeness theorem employs sets of clauses no clause (except ~A itself) containing ~A in proofs of the subgoals. Thus, no π reduction operation can result in a subgoal of A.

2. No goal may have as a subgoal (immediate or otherwise) a complementary opened goal. This is equivalent to the model

elimination admissability rule that no chain with an A¬literal followed by a complementary A¬literal needs to be retained.

The justification for this in the * procedure is that the lower goal could have been * reduced by the negated higher goal. This is assumed to be done in some alternate deduction.

3. No opened goal may be an instance of an original premise. This rule has no counterpart in the formal definition of the model elimination procedure (all admissability rules refer to single chains and take no notice of the input chains), but was recognized and successfully used by Fleisig et al [12] and in PSTP.

The justification for this in the * procedure is that the opened goal could have been * reduced by the original premise. This is assumed to be done in some alternated deduction.

4. No goal may be an instance of the negation of an original premise. This rule, though fairly obvious, was apparently previously unrecognized, and has only recently been incorporated in PSTP.

The justification for this in the * procedure is that (examining the proof of the completeness theorem) fully unit subsumed clauses are used. Thus, if A is an original premise, the completeness theorem employs sets of clauses no clause (except A itself) containing A. Thus, no * reduction operation can result in a subgoal ~A. (This is an extension of rule 1 to axioms as well as negated higher goals.)

Note. Rule 4 has the following interpretation in terms of the

model elimination procedure. No chain with an A-literal or B-literal which is an instance of a unit axiom needs to be retained. If the literal occurs to the right of the first A-literal of the chain, it must have come from extension by some axiom. But this (instance of) the axiom is subsumed by the unit axiom, and is thus unnecessary to a proof (by the completeness of the model elimination procedure). If the literal occurs to the left of all the A-literals of the chain, then the (instance of the) top chain is subsumed by the unit axiom. Thus, if the top chain is refutable, so is the unit axiom, and there exists an alternate refutation (although with different top chain).

2.12. Multiple world states

We have been exclusively conerned up to this point with reasoning within a single world state. That is, we have considered only the case of establishing a theorem in the presence of some axioms without considering the possibility of the set of axioms being impermanent and changeable.

Yet an important feature of artificial intelligence languages such as PLANNER and QA4/QLISP is their ability to represent different world states by adding or deleting axioms. Thus we must consider the effect of this expanded field of operation on our completeness claims for the π procedure which we propose as a model for a complete inference system for artificial intelligence languages.

As demonstrated by Green [17], the multiple world state situation can be handled directly within the first order predicate calculus (and thus by the unmodified * procedure) by the use of state variables in state-dependent predicates.

We will here distinguish three kinds of reasoning: reasoning within a world state, reasoning between world states, and reasoning about world states.

Reasoning within a world states corresponds to the conventional theorem proving problem and is directly handled by the * procedure.

By reasoning between world states, we mean the typical operation of PLANNER and QA4/QLISP by which, in order to achieve a goal, some operation is performed which results in a different state by the addition or removal of axioms. For example, the goal [inroom person1 , room1] may be achieved by the walk operator which removes [inroom person1 , x] for all x and adds [inroom person1 , room1]. If this is only one component of a larger task, the solution of the larger task would continue in this altered world state. Reasoning between world states includes explicit transformations, usually corresponding to physical actions, which transform one state to another.

Aside from the state modification which takes place when assertions are added to or deleted from the data base as a result of successful application of a world state transformation, the principal effect of reasoning between world states is the absence of a need to assert the negation of a goal (to invoke the

contradiction mechanism) if the goal is to be achieved (producing a new world state) rather than merely inferred (proved in the current world state). The reason for this is that the predicate calculus state variable description of the state to be achieved cannot be unified with the state variable of any predecessor state. Hence, no complemented lower goal can be discovered.

This fact is further justification for the already useful distinction between the achieve and infer goal functions of Popler 1.5 [10]. The infer type goal tries to prove the goal in the present world state and would need to (using the π procedure) check for complementarity of higher and lower subgoals. The achieve type goal which tries to produce the goal would not need to perform this checking.

The third kind of reasoning with world states, reasoning about world staes, may involve the creation of hypothetical world states which are then reasoned about, or determining relationships between different world states. Hewitt [22] suggests the example of comparing the weight of Neil Armstrong on the earth ($context_4$) and on the moon ($context_5$). Present artificial intelligence languages are rather weak in conveniently allowing such comparisons, and Hewitt suggests the alternative of, rather than storing data in contexts, putting explicit context tags (called situational tags) in the data. This is clearly a general and complete solution to the problem of relating the data to the world state.

A curiosity unmentioned by Hewitt is that his use of

situational tags (though presented with different rhetoric and motivated by the example of QA4/QLISP contexts) constitutes a return to the concept of Green's predicate calculus state variables from which situational tags are essentially indistinguishable. The treatment of situational tags in the fashion of predicate calculus state variables permits the ⊬ procedure to be used for this reasoning about world states.

2.13. Equality

In analogy to the MESON procedure for equality, there is a ⊬ procedure with an equality inference rule. In addition to the ⊬ reduction inference rule, the ⊬ procedure for equality includes a single additional inference rule, the ⊬ substitution rule. Substitution from a higher goal (A-literal) is handled by ⊬ substitution from a locally asserted negated higher goal just as reduction from a higher goal is handled by ⊬ reduction from a locally asserted negated higher goal.

One change that is made to the organization of the ⊬ procedure in preparation for use of ⊬ substitution is that now the negated goal is asserted before any inference operation is attempted on the goal, not after as was possible when only ⊬ reduction was used. This is to allow for substitution of a goal to itself (ME-P p-reduction to self).

The following defines the ⊬ substitution inference rule.

Definition 2.31. $B_1,\ldots,B_p \vdash_{\pi} C$ if

B_k or $\neg C$ is $A_1 \wedge \ldots \wedge A_m \rightarrow C'$ $(m \geq 0)$

and $B_1,\ldots,B_p,\sim C \vdash_{\pi} D$ (defined below)

and $B_1,\ldots,B_p,\sim C \vdash_{\pi} A_1$

:

and $B_1,\ldots,B_p,\sim C \vdash_{\pi} A_i$

:

and $B_1,\ldots,B_p,\sim C \vdash_{\pi} A_m$ for some k ($1 \leq k \leq p$).

Case I. (π substitution from assertion) If C' is of the form a=b or b=a and C has an instance of term b then D is C with a single instance of b replaced by a.

Case II. (π substitution to assertion) If C is of the form $a \neq b$ or $b \neq a$ and C' has an instance of term b then D is ~C' with a single instance of b replaced by a.

The following theorem extends our previous soundness proof for the π procedure without equality to the π procedure with equality.

Theorem 2.6. The π procedure with equality is sound, i.e., if $B_1,\ldots,B_p \vdash_{\pi} C$ then $B_1,\ldots,B_p \vdash C$.

Proof. We will perform induction on the number of applications n of the π reduction and π substitution inference rules used in a π procedure proof.

n=1. Assume $B_1,\ldots,B_p \vdash_{\pi} C$ with only one application of an inference rule in the proof. Since only one application of an inference rule is required, π reduction must be used with B_k being C itself. Otherwise, additional inferences would be required to prove $A_1,\ldots,A_m$ in the case of a non-premise

application of π reduction, or to prove $A_1,\ldots,A_m$ and D in the case of π substitution.

n>1. Assume the π procedure is sound for all proofs with fewer than n applications of inference rules. Prove it is sound for n. Assume $B_1,\ldots,B_p \vdash_\pi C$ with n applications of inference rules.

Case 1. The case where the π reduction rule is the top inference rule application is already proved for the soundness theorem for the π procedure without equality.

Case 2. The case where the π substitution rule is the top inference rule application employs the subproofs $B_1,\ldots,B_p,\sim C \vdash_\pi D$, $B_1,\ldots,B_p,\sim C \vdash_\pi A_1$, ..., $B_1,\ldots,B_p,\sim C \vdash_\pi A_i$, ..., and $B_1,\ldots,B_p,\sim C \vdash_\pi A_m$ where B_k or $\sim C$ is $A_1 \wedge \ldots \wedge A_m \rightarrow C'$ for some k ($1 \leq k \leq p$). By the induction hypothesis, since each of these subproofs has fewer than n applications of inference rules, $B_1,\ldots,B_p,\sim C \vdash D$, $B_1,\ldots,B_p,\sim C \vdash A_1$, ..., $B_1,\ldots,B_p,\sim C \vdash A_i$, ..., and $B_1,\ldots,B_p,\sim C \vdash A_m$. Therefore, $B_1,\ldots B_p \sim C \vdash D \wedge A_1 \wedge \ldots \wedge A_m$ and, by the deduction theorem, $B_1,\ldots B_p \vdash \sim C \rightarrow D \wedge A_1 \wedge \ldots \wedge A_m$ and $B_1,\ldots,B_p \vdash \sim(D \wedge A_1 \wedge \ldots \wedge A_m) \rightarrow C$.

Case 2a. Consider first the case where B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C'$, i.e., the case where C' is not $\sim C$. Then $B_1,\ldots B_p \vdash A_1 \wedge \ldots \wedge A_m \rightarrow C'$. From this plus the previous fact that $B_1,\ldots,B_p \vdash \sim(D \wedge A_1 \wedge \ldots \wedge A_m) \rightarrow C$ we can infer that $B_1,\ldots,B_p \vdash \neg(D \wedge C') \rightarrow C$. There are two possible cases corresponding to cases I and II of the π substitution inference rule to be considered: (1) C' is a=b or b=a, C is C(b), and D is C(a), and (2) C' is C(b), C is

$a \neq b$ or $b \neq a$, and D is $\sim C(a)$. We will construct a truth table for the values of C, D, C', and $\neg(D \wedge C') \rightarrow C$.

C	C'	D	$\neg(D \wedge C') \rightarrow C$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

We need only consider the lines in which $\neg(D \wedge C') \rightarrow C$ is true since we know this to be a theorem provable from $B_1,\ldots,B_p$. In each such line except the fifth, C has value true and is thus provable from $B_1,\ldots,B_p$ satisfying the requirements of the theorem. In case (1), line 5 states the following are true: $\neg C(b)$, $C(a)$, and $a=b$ which is inconsistent with the standard interpretation of equality. In case (2), line 5 states the following are true: $a=b$, $\sim C(a)$, and $C(b)$ which is likewise inconsistent with the standard interpretation of equality. Thus, C is provable everywhere $\neg(D \wedge C') \rightarrow C$ is satisfying the statement of the theorem.

Case 2b. Now consider the case where C' is $\sim C$. If $B_1,\ldots,B_p$ are inconsistent, then $B_1,\ldots,B_p \vdash C$ immediately. If $B_1,\ldots,B_p$ are consistent, and we have proved $B_1,\ldots,B_p \vdash \sim(D \wedge A_1 \wedge \ldots \wedge$

$A_m) \rightarrow C$ which reduces to $B_1,\ldots,B_p \vdash \sim D \rightarrow C$, or $B_1,\ldots,B_p \vdash D \vee C$. If D is false, then D must be true, and $B_1,\ldots,B_p \vdash D$. If D is true, assume C is false. Then, by substitution of equals for equals, since $\sim C$ is of the form a=b and D is derived from C by substituting a for b, C and D are equivalent. Contradiction. Therefore $B_1,\ldots,B_p \vdash C$. □

The * procedure for equality can be seen to be complete by analogy to the MESON procedure for equality, and ME-P from which both are derived.

2.13.1. Marking and other restrictions. In addition to the problem of equality inference rules producing a great many consequences, there is the additional problem that they tend to produce the same consequences in many different ways, representing different permutations in the order in which operations are performed. Such multiple derivations are often eliminated in the case of theorem proving without equality by use of ordering restrictions. Some of the restrictions we present here for equality have the flavor of extending ordering restrictions to equality inference rules.

Several completeness preserving restrictions on the use of * substitution can be used to eliminate some of the multiple derivations of terms.

Restriction 1. The first restriction consists of requiring * substitution from an axiom (an assertion in the original set of assertions as opposed to an assertion which is the negation of a higher goal) to substitute the first term for the second and to

never substitute the second for the first. Schematically, the axiom a=b or $A_1 \wedge \ldots \wedge A_m \rightarrow a=b$ will be permitted to substitute (an instance of) term a for term b in some literal, but not term b for term a. Clearly, preservation of completeness requires that both substitution possibilities be available. This is done by including in the axioms both a=b and b=a if a differs from b. Since, in general, both variants are required for completeness in every case where a differs from b, this restriction does not decrease the number of operations that can be performed except in the case where a is identical to b. In the latter case, * substitution by a=a (e.g., * substitution by functionally reflexive axiom f(xy)=f(xy)) will produce fewer identical results since it will now only try to substitute the left f(xy) for the right f(xy) and not vice versa. Aside from this reduction in the number of operations in the case where a is identical to b, the principal advantage of this restriction is that the two substitution possibilities (a for b, and b for a) can be distinguished and differentially employed in the search for a solution. For example, in group theory problems, the axiom x=f(xe) (x equals x times the right identity e) can be used in two ways: replacing a term of the form f(ae) by a or replacing a term of the form a by f(ae) for arbitrary a. Since the first possibility is a case of simplifying the expression, and the second makes the expression more complex, the first usage of the axiom x=f(xe) is generally preferred to the second. The two possibilities are readily distinguished using this restriction by

writing both axioms x=f(xe) and f(xe)=x. Since systems such as PSTP operate on chains (clauses, axioms, etc.) more than on literals or terms, the alternative substitution possibilities are very readily distinguished by using this restriction.

The choice of whether axiom a=b is to represent the substitution of a for b, or b for a, is clearly an arbitrary one since either convention would work equally well. The convention used (namely, that a=b denotes the substitution of a for b) was chosen for its mnemonic value. Just as * reduction by A → B establishes as a subgoal of B the goal A, a=b establishes as a subgoal of a literal containing b a goal containing a. Thus, a=b is rather like A → B only it operates on terms rather than literals.

Restriction 2. The second restriction, as opposed to the first, has a substantial impact on reducing the number of alternate derivations of terms by * substitution. This restriction results from the observation that structure is introduced into a term by two different methods. The first method is * substitution from a functionally reflexive axiom. This is the standard structure introducing operation, required for completeness as shown by the theorem lifting completeness of the * procedure with equality from the ground level to the general level. The second method involves * substitution from an axiom which is not a functionally reflexive axiom which instantiates a variable of the term being substituted into.

An example of each method involves derivations from the term

f(xa). ▾ substitution by the functionally reflexive axiom f(xy)=f(xy) results in f(f(xy)a), replacing the x of the original term by f(xy) introducing the additional structure f(...). ▾ substitution by the associativity axiom f(xf(yz))=f(f(xy)z) results in f(xf(ya)), replacing the x of the original term by f(xy) introducing the additional structure f(...) (just like ▾ substitution by the functionally reflexive axiom) and then rearranging (parentheses of) the term.

Note that the term f(xf(ya)) derived by ▾ substitution by the associativity axiom is also derivable by first ▾ substituting by the functionally reflexive axiom and then ▾ substituting by the associativity axiom. Thus, there are two alternate derivations of f(xf(ya)).

The purpose of this restriction is to eliminate one of these derivations by restricting structure introducing operations to the functionally reflexive axioms.

The restriction is that, except in the case of ▾ substitution by a functionally reflexive axiom, no variable of the term being ▾ substituted into may be instantiated by the ▾ substitution operation to a term which is neither a variable nor a constant. Any ▾ substitution operation which did so instantiate a variable must be performed on the modified term after a sequence of ▾ substitution operations by functionally reflexive axioms.

As well as decreasing the number of alternative derivations of a term, this restriction like the previous one has some heuristic value since it makes operations which introduce structure (▾

substitution from functionally reflexive axioms) and operations which, for example, rearrange structure such as associativity and commutativity readily distinguishable.

Restriction 3. The third restriction, the marking restriction, is the most sophisticated of the 3 restrictions presented here. It is an attempt to achieve the same effect of reducing the number of alternate derivations when equality inference rules are used as is achieved by chain or clause ordering rules in the absence of equality.

Chain or clause ordering rules impose restrictions on the order in which literals may be resolved upon, depending on their lexicographic value (A-ordering) or their inherited position (C-ordering, used in the model elimination derived procedures). The marking restriction imposes an ordering restriction on substitution operations on terms.

Carelessly stated, the marking restriction prohibits a substitution operation on a term lexically to the left of a term previously substituted.

An illustration of the potential effectiveness is the following.

Example 2.19.

1. Pabc
2. a=d
3. b=e
4. c=f
5. Pdef theorem

There are 6 proofs of the theorem, varying only in the order in which substitution from the axioms is performed.

6.1. [Pdef] Paef * substitution from 2
7.1. [Pdef] [Paef] Pabf * substitution from 3

```
8.1. [Pdef] [Paef] [Pabf] Pabc   ¤ substitution from 4
9.1. ø                           ¤ reduction by 1

6.2. [Pdef] Paef                 ¤ substitution from 2
7.2. [Pdef] [Paef] Paec          ¤ substitution from 4
8.2. [Pdef] [Paef] [Paec] Pabc   ¤ substitution from 3
9.2. ø                           ¤ reduction by 1

6.3. [Pdef] Pdbf                 ¤ substitution from 3
7.3. [Pdef] [Pdbf] Pabf          ¤ substitution from 2
8.3. [Pdef] [Pdbf] [Pabf] Pabc   ¤ substitution from 4
9.3. ø                           ¤ reduction by 1

6.4. [Pdef] Pdbf                 ¤ substitution from 3
7.4. [Pdef] [Pdbf] Pdbc          ¤ substitution from 4
8.4. [Pdef] [Pdbf] [Pdbc] Pabc   ¤ substitution from 2
9.4. ø                           ¤ reduction by 1

6.5. [Pdef] Pdec                 ¤ substitution from 4
7.5. [Pdef] [Pdec] Paec          ¤ substitution from 2
8.5. [Pdef] [Pdec] [Paec] Pabc   ¤ substitution from 3
9.5. ø                           ¤ reduction by 1

6.6. [Pdef] Pdec                 ¤ substitution from 4
7.6. [Pdef] [Pdec] Pdbc          ¤ substitution from 3
8.6. [Pdef] [Pdec] [Pdbc] Pabc   ¤ substitution from 2
9.6. ø                           ¤ reduction by 1
```

Thus, there are 6 occurrences of the empty chain at level 4 and 24 derived chains in all (disregarding the possibility of subsumption being used to eliminate identities).

If the marking restriction is used, then the only totally acceptable proof is the first one since it is the only one that substitutes for the terms d,e,f in strict left to right order. Formation of the following chains is prohibited by the marking restriction since each requires substitution for a term to the left of a previously substituted term: 8.2, 7.3, 8.4, 7.5, 7.6. Thus, at level 4 there is only 1 occurrence of the empty chain and 11 derived chains in all, a 54% reduction.

We will now present the marking restriction in greater detail.

1. (marked term usage restriction) Each term in the last literal of a chain may be marked. No π substitution operation from a higher goal (A-literal) or from an axiom can be performed on a marked term directly. For example, if c=d is an axiom and d' a term of the last literal of the chain where d and d' are unifiable, no marked occurrence of d' in the literal can be π substituted by c=d. Marked occurrence of d' may be used in π substitution operations from higher goals or axioms if they are subterms of the unmarked term being matched in the π substitution operation.

2. (acquisition of marks) Each term totally to the left of the term matched in a π substitution operation from a higher goal or axiom is marked. If a term is marked, so are all its substerms.

3. (retention of marks of unmatched terms) Marks are retained by all marked terms not matched in a π substitution operation. If the marked term is a variable which is instantiated by the π substitution operation, its value is marked at the marked occurrence.

4.1. (retention of marks of matched terms) The conservative approach is to eliminate all marks of terms matched in a π substitution operation from higher goal or axiom. This conservative marking restriction still imposes a left to right order in performing π substitution operations on disjoint terms, but wipes out all marks when a superterm is π substituted. An illustration of why this approach is too conservative relates to the use of commutative and similar axioms. Every term

immediately preceded by "!" or contained in a term immediately preceded by "!" is marked.

Example 2.20.

```
1. Pf(ab)
2. a=c
3. b=d
4. f(xy)=f(yx)
5. Pf(dc)                theorem
6. Pf(!da)               * substitute from 2
7. Pf(ad)                * substitute from 4, destroy marks
8. Pf(!ab)               * substitute from 3
9. ø                     * reduce by 1
```

A consequence of destroying the marks in chain 7 using the conservative approach is that the marking restriction had no effect in reducing the number of possible proofs. It is desirable to strengthen the marking restriction to allow for inheritance of marks through * substitution operations.

We illustrate this idea using the same example. Here, terms which are marked when matched to the variables y and x of f(yx) of axiom 4 retain their marks when substituted in f(xy).

Example 2.21.

```
5. Pf(dc)                theorem
6. Pf(!da)               * substitute from 2
7. Pf(a!d)               * substitute from 4, marks inherited
```

Chain 8 can no longer be derived since it would required * substitution on the marked term d. Remaining valid deductions are

```
5. Pf(dc)                theorem
6. Pf(bc)                * substitute from 3
7. Pf(cb)                * substitute from 4
8. Pf(ab)                * substitute from 2
9. ø                     * reduce by 1

5. Pf(dc)                theorem
6. Pf(cd)                * substitute from 4
7. Pf(ad)                * substitute from 2
8. Pf(!ab)               * substitute from 3
```

9. ø ¤ reduce by 1

Inheritance of marks is not always so straightforward as is suggested by the example using the commutative axiom. Freely inherited marks may result in blocking all proofs.

Example 2.22.

1. Pg(de)
2. c=b
3. g(xx)=f(xc)
4. d=a
5. e=a
6. Pf(ab) theorem

7.1. Pf(!ac) ¤ substitute from 2
7.2. Pg(!a!a) ¤ substitute from 3
(blocked)

8.1. Pf(db) ¤ substitute from 4
8.2. Pf(!dc) ¤ substitute from 2
8.3. Pg(!d!d) ¤ substitute from 3
(blocked)

9.1. Pf(eb) ¤ substitute from 5
9.2. Pf(!ec) ¤ substitute from 2
9.3. Pg(!e!e) ¤ substitute from 3
(blocked)

The problem here is that b must be ¤ substituted to c in order to apply necessary axiom 3. a, or whatever a has already been ¤ substituted to, is then marked. When two copies of it are formed, both are marked prohibiting the ¤ substitution of each, eliminating any possibility of ¤ reduction by 1.

4.2. (better retention of marks of matched terms) It is apparent that marks may be validly retained when there is only one occurrence of the variable in the left side of the equality (such as in the case of the commutative axiom). Thus, marks may be transmitted through each variable of the equality for which there is only one ocurrence of the variable in the left side.

The problem of the most recent example refuting the general transmission of marks through variables of the equality can be overcome by specifying that marks may be transmitted through at most one occurrence of each variable in the left side of the equality. A consistent convention (such as transmitting marks only through the leftmost occurrence in the left side) must be adopted. Thus, in the previous example, only one of the two a's of 7.2, one of the two d's of 8.3, and one of the two e's of 9.3 would be marked. This would allow (assuming the leftmost variable occurrence retains marks) 8.3 (by # substituting first a=d, then e=a) to be extended to a complete proof. Both other deductions remain blocked. This lengthened proof (longer than the shortest proof with no marking restriction) can be shortened by specifying no variable occurrence in the left side of the equality transmits marks if there is more than one occurrence of the variable in the left side. However, the cost of discovering such lengthened proofs should generally be overcome by the lowered cost of searching for a proof provided by the stronger marking restriction. Note however that this stronger marking restriction may require the use of equality substitutions (a=d in the example) not otherwise necessary and more equality literals may need to be included if restriction 1 is used. A more general comment is that transmission of marks through one of the n>1 variable occurrences of the left side requires the symmetry of substitution, i.e., if a can be substituted to b then b can be substituted to a.

An inefficient form of the marking restriction was implemented in an earlier version of PSTP by Don Cohen in June 1975. It was tested on an equality based formulation of the group theory problem: if the square of every element is the identity element then the group is Abelian. Several equalities are given twice, once in reversed order, since restriction 1 was used.

Example 2.23.

1. c=f(ab)
2. e=f(xx)
3. x=f(xe)
4. x=f(ex)
5. f(xx)=e
6. f(xe)=x
7. f(ex)=x
8. f(xf(yz))=f(f(xy)z)
9. f(f(xy)z)=f(xf(yz))
10. c=f(ba) theorem

* substitution from the axioms was performed exhaustively up to level 2. The A-literal identical to following B-literal chain deletion rule was used. The number of chains retained with and without the marking restriction is given in the table below.

	Level 0	Level 1	Level 2
without marking	1	8	110
with marking	1	8	94

The 1 level 0 chain is the theorem. The marking restriction makes no difference at level 1 since the theorem has no marks to restrict the number of * substitution operations which can be performed. Marking only has an effect at the level 2 and above. The marking restriction resulted in a 15% reduction in the number of chains at level 2.

In addition, several chains at level 2 were * substituted with and without the marking restriction. The following table

expresses the number of chains resulting ▼ substituting each of the specified level 2 chains.

			without	with
[c=f(ba)]	[f(ec)=f(ba)]	f(f(x1x1)c)=f(ba)	33	33
[c=f(ba)]	[f(ec)=f(ba)]	!f(ec)=f(!bf(ae))	20	10
[c=f(ba)]	[f(ec)=f(ba)]	!f(ec)=f(f(be)a)	20	12
[c=f(ba)]	[f(ec)=f(ba)]	!f(ec)=f(f(ba)e)	20	12
[c=f(ba)]	[f(ec)=f(ba)]	f(!ef(ce))=f(ba)	20	17
[c=f(ba)]	[f(ec)=f(ba)]	f(f(ee)c)=f(ba)	19	19
[c=f(ba)]	[f(ec)=f(ba)]	f(f(ec)e)=f(ba)	20	20
[c=f(ba)]	[f(ec)=f(ba)]	!f(ec)=f(!bf(ea))	20	10
[c=f(ba)]	[f(ec)=f(ba)]	!f(ec)=f(f(eb)a)	20	12
[c=f(ba)]	[f(ec)=f(ba)]	!f(ec)=f(ef(ba))	20	12

The marking restriction resulted in a 0% (when no marks were present) to 50% reduction in the number of chains with a mean of 26% reduction. The latter figure, which substantially exceeds the 15% reduction at level 2, confirms our expectation that the marking restriction becomes more effective at higher levels due to the increased complexity of the term being ▼ substituted.

The remaining high branching factor for the ▼ substitution operation even in the presence of the marking restriction indicates that although it may substantially reduce the number of generated chains, it is not a panacea to the general problem of too many chains being generated.

A reasonable question is whether the marking restriction can be extended to retention of marks during other inference (e.g., ▼ reduction by an axiom) operations. The proposed extension would (where applicable) cause a marked term to remain marked in every location where it was substituted. (Presently, the only operations which preserve marks are ▼ substitution from an axiom and ▼ substitution from a higher negated goal from which the marked literal is not itself derived solely by ▼ substitution.)

The following example demonstrates the incompleteness of the proposed extension to the marking restriction.

Example 2.24.

1. b=a
2. Qx ∧ Rx → Pxb
3. Rd
4. c=d
5. Qc
6. Pxa — theorem
7. [Pxa] P!xb — π substitution from 1 (x is marked)
8. [Pxa] [Pxb] Q!x R!x — π reduction by 2 (x's retain marks according to proposed extension)
9. [Pda] [Pdb] Q!d — π reduction by 3
10. [Pda] [Pdb] [Qd] Qc — π reduction by 4
11. ∅ — π reduction by 5

The reason this proof is invalid is that chain 9 cannot be formed when the marking restriction is applied since d of Qd is marked.

There are alternate proofs, i.e., where x is π substituted to c or d before a is π substituted to b. These other proofs are also blocked since, after a is π substituted to b, the value of x (c or d) is marked yet must be π substituted again to handle π reduction by both Qc and Rd.

We will next explore an alternate method for controlling this excessive generation of chains: demodulation, which is unfortunately not compatible with our marking restriction since they are so totally different in philosophy and application.

2.13.2. Demodulation. Demodulation (developed by Wos et al [54]) offers a partial solution to the explosive growth in the number of partial deductions introduced by use of the equality predicate.

We will first introduce some definitions.

Definition 2.32. A modulation list is a list $a_1=b_1$, ..., $a_n=b_n$ of equality literals.

Definition 2.33. If E is an expression containing an instance of term b, and E' is E with a single instance of b replaced by a, then E' is an immediate modulant of E relative to any modulation list containing a=b.

Definition 2.34. Expression E' is a k-modulant of expression E relative to modulation list D if there exists a sequence of immediate modulants E_1, ..., E_n relative to D such that $E_1=E$, $E_n=E'$, and $n\leq k$. Note that E is a k-modulant of E.

Definition 2.35. A demodulation list is a modulation list such that for each equality literal a=b in the list, a has fewer symbols (is simpler) than b.

We could relax the restriction on a having strictly fewer symbols than b by adopting complexity measures similar to those used in complete sets of reductions. Using appropriately defined complexity measures, f(f(xy)z) could be replaced by f(xf(yz)) and g(f(xy)) could be replaced by f(g(y)g(x)). If complete sets of reductions are used, we are also assured that demodulation is deterministic, a desirable property.

We could also allow equalities with an equal number of symbols on each side (such as the commutativity axiom f(xy)=f(yx)) and redefine demodulation to select one of the set of all demodulants with minimal complexity.

Definition 2.36. Expression E' is a demodulant of expression E relative to demodulation list D if it is a k-modulant of E relative to D for some k and has no immediate modulant relative to D.

In a resolution theorem proving system, demodulation is used to produce the demodulated form of each newly derived clause. The demodulation list is a list of equality consequences of the set of axioms. When the literals of two clauses which are being resolved are unified against each other, provision must be made for the possibility that each is the demodulated form of more complex literals which actually must be unified. Hence, k-modulants of each of the literals must be unified where k is generally unbounded. The demodulated literal is regarded as representing all literals from which it could be produced by demodulation.

In contrast to other equality restrictions such as marking, the use of demodulation is not restricted to cases in which equality inference rules are being used. Demodulation is often used to simplify terms even when resolution is the sole inference system.

There are several reasons for the proposed and experimentally noted success of demodulation:

1. demodulation retains terms in their simplest, most likely to be useful form,
2. reducing the variety of terms by demodulation often results in extra subsumption operations being successful,

3. reducing the variety of terms by demodulation may result in extra tautology or other similar deletion operations being successful, and

4. demodulation is often implemented in an incomplete system (terms are not fully k-modulated back upward in complexity in matching) resulting in fewer clauses being produced.

The great success of demodulation, often permitting the solution of problems not otherwise solvable, conceals the fourth reason almost totally.

Demodulation has generally been presented with little formal justification (perhaps because though it is not obvious that it works it is at least intuitive). However, it is necessary for us to justify its use a little more formally than usual when applying it to the π procedure. We need to verify that the higher goal complementary to lower subgoal deletion rule is still valid for demodulated deductions. It is not obvious that it is since it is likely that different occurrences of the same demodulant really refer to different literals. Eliminating deductions with complementary demodulants may eliminate all proofs.

Theorem 2.7. Substitution by implied equivalence preserves consistency for ground formulas. If $S \vdash A \leftrightarrow B$ then $S\{B \leftarrow A\}$ (S with B uniformly replaced by A) is consistent (inconsistent) if S is consistent (inconsistent) where S is a ground formula, A and B are ground atomic formulas.

Proof.

Case 1. S is consistent. Then S has a model M. Since $S \vdash A \leftrightarrow B$ by hypothesis, M assigns the same truth value to A and B. The substitution of A for B in S will therefore still have model M. Therefore, since it has a model, $S\{B \leftarrow A\}$ is consistent.

Case 2. S is inconsistent. Then S by definition has no model. Hence, it also has no model which assigns the same truth value to A and B, and the substitution of A for B in S will not result in a consistent formula. Therefore, $S\{B \leftarrow A\}$ is inconsistent. □

To facilitate the proof of the following theorem, we will make one plausible assumption about demodulation--that demodulation is deterministic and that the same formula will not be demodulated in two different ways in the same proof.

Theorem 2.8. Demodulation is sound and complete for the * procedure.

Proof. Let S be an inconsistent set of clauses. Let C be any essential clause, i.e., S-C is consistent. Then by Herbrand's theorem there is a minimally inconsistent set of ground clauses S' each element of which is an instance of a clause of S and some clause C' contained in S' is an instance of C.

Let $S \vdash A_1 \leftrightarrow B_1$, $S \vdash A_2 \leftrightarrow B_2$, ..., $S \vdash A_n \leftrightarrow B_n$. Then let S_1' be $S'\{A_1 \leftarrow B_1\}$, S_2' be $S_1\{A_2 \leftarrow B_2\}$, ..., S_n' be $S_{n-1}\{A_n \leftarrow B_n\}$. Similarly, let C_n' be $(\ldots((C'\{A_1 \leftarrow B_1\})\{A_2 \leftarrow B_2\})\ldots)\{A_n \leftarrow B_n\}$. These are all atomic formula replacements at the ground level effected by a demodulation list.

Then by the previous theorem S_n' is inconsistent and $S_n' - C_n'$ is

consistent. This satisfies the requirements for application of the π procedure. Hence, there is a π proof of $\neg C'_n$ from S'_n.

This π proof can be altered to be a π procedure with demodulation proof of $\neg C'$ from S' by incorporating the $A_1 \leftrightarrow B_1$, ..., $A_n \leftrightarrow B_n$ equivalences in the literal matching process. The resulting proof does not violate the higher goal complementary to lower goal subgoal restriction (if the proof of C'_n does not) since (by the assumption of demodulation being deterministic) identical goals in the proof of $\neg C'$ must correspond to identical goals in the proof of $\neg C'_n$; when transforming the proof from one proving $\neg C'_n$ to one proving $\neg C'$, the atomic formulas in the proof can only get more distinct--no new identities are produced.

The proof of $\neg C'$ from S' can easily be lifted to a proof of $\neg C$ from S by the π procedure with demodulation.

The π procedure with demodulation is complete by the demonstration of the existence of this proof. It is sound since substitution by implied equivalences (of which demodulation is an example) is consistency preserving, and by the soundness of the π procedure applied to S'_n. □

2.14. Use of models

The use of models should be an important component of deductive systems. They can substantially reduce computational effort by indicating which goals cannot possibly be proved.

Since pairs of individually complete theorem proving

refinements are not necessarily complete when used in combination, and it is well known that the combined semantic and linear resolution refinements are incomplete, it is worth considering whether there is a usage of models which is complete when used with the π or MESON procedures.

We will start by describing the most successful use of models in deduction to date, the Geometry-Theorem Proving Machine. Then, after giving a brief taxonomy of models, we will indicate why direct application of Geometry-Theorem Proving Machine model ideas will not work for the π or MESON procedures.

It will be shown that there are circumstances in which it is necessary to attempt to prove subgoals in the π or MESON procedures which are not only false in a model, but which are actually disprovable from the axioms. We will then present a possible solution in the form of a modified π or MESON procedure.

2.14.1. Geometry-Theorem Proving Machine. Gelernter et al.'s Geometry-Theorem Proving Machine (GTM) [13,14,15,16] is a theorem proving program for Euclidean plane geometry. The deductive portion of GTM functions entirely in the (decidable, see Tarski [50]) domain of Euclidean plane geometry which excludes constructions. An additional facility in GTM permits constructions when GTM cannot find a solution without one.

GTM is a pure problem reduction method based theorem proving program. It is presented a list of hypotheses and a conclusion expressed in terms of geometric predicates and points. The latter are, in a more theoretical sense, Skolem constants. GTM

then works backward from the conclusion to the hypotheses and known identities using theorems of geometry. The primary heuristic in GTM is the presence of a model to reject false goals. The model takes the form of a geometric diagram from which measurements are taken and is constructed so as to satisfy the premises of the theorem to be proved.

The notation employed by Gelernter [13] in describing the operation of GTM is somewhat different from that which is currently used to describe problem reduction method systems. In his terminology, the application of a theorem to a subgoal generates a single subgoal. This subgoal, however, may be a conjunctive set of terms none of which separately is considered to be a subgoal. Solution of each term constitutes solution of the subgoal. The following description of GTM will use a more contemporary representation for subgoals and dismiss this notational distinction between terms and subgoals.

In GTM, immediate subgoals of the current goal are generated in breadth first fashion. More precisely, in order to solve a single subgoal, all applicable theorems are attempted in parallel. Each successful application of a theorem (the consequent of the theorem matches the subgoal) results in the formation of a conjunctive set of subgoals whose successful simultaneous solution constitutes a solution for the original subgoal.

GTM does not allow unbound variables in its problem solving graph. Thus, the next step is to replace each set of conjunctive

subgoals having free variables by all alternative conjunctive sets with variables replaced by points specified in the problem statement.

Subgoals are then subjected to a test to determine whether they are true in the diagram. Sets of subgoals containing subgoals false in the diagram are rejected. (Actually, the two steps of (1) instantiating variables by points in all possible ways and (2) rejecting conjunctive sets of subgoals containing a false subgoal can be performed partially as one step with the analysis of the diagram actually proposing points rather than taking all instantiations of the subgoals and testing each one.)

Generated sets of subgoals are then subjected to several tests to determine if they should be added to the subgoal tree ("problem solving graph"). These rules are somewhat heuristic in nature and could be improved by the use of model elimination or MESON procedure acceptability and subsumption rules.

If B is a subgoal of A and has no partner goals, no other conjunctive set of goals containing either B or a symmetric variant of B will be added to the problem solving graph as subgoals of A. The solution of the original B in the problem solving graph is implied by the solution of any of the other sets of subgoals containing B and is alone sufficient to solve A. This is really just a subsumption condition.

No conjunctive set of subgoals will be added to the problem solving graph if this would result in a subgoal being identical to or a symmetric variant of an ancestor goal. This is

essentially equivalent to the model elimination or MESON procedure statement that chains with an A-literal or higher goal followed by an identical B-literal or lower goal can be discarded without loss of completeness.

A third rule eliminates the addition of a subgoal without partner goals that is identical to or a symmetric variant of a subgoal without partner goals that is a subgoal of an ancestor of the proposed added goal. In linear MESON procedure terms, the restriction is that if chain [A] B is part of the problem solving graph, chains [A] $[C_1]$ ... $[C_n]$ B or [A] $[C_1]$ ... $[C_n]$ B' will not be added (where B' is a symmetric variant of B). This is, once again, a subsumption rule. It would be a valid inference rule in model elimination or the MESON procedure in the absence of reduction (which is not present in GTM) since the presence or absence of A-literals is then irrelevant to finding a solution.

The final rule specifies the rejection in some cases of conjunctive sets of subgoals that are supersets of other conjunctive sets of subgoals. If the larger is descended from a member of the smaller, the larger is rejected; if the smaller is descended from the larger, both are retained. This part of the rule is again similar to the A-literal identical B-literal rule in model elimination or the MESON procedure. If one is not descended from the other, it is retained only if it has fewer elements. This then is essentially another subsumption rule.

A conjunctive set of subgoals is rejected if any of its elements is labeled as stuck. Subgoals become stuck by exhausting all alternative proofs.

If the subgoal is acceptable according to the above criteria, it is then checked to see if it is labeled as already established or (in certain cases, notatably identities) if there exists a short (one step) solution.

The search strategy dictates that solution of a conjunctive set of subgoals will cause GTM to start working on the next unsolved ancestor partner goal.

When forced to choose a subgoal to work on, candidates are compared to the hypotheses with respect to points, connectives, and predicates and some distance function computed. The subgoal distance score is then expanded. (Note: this evaluation function differs from the admissable measures of Nilsson [35] as mentioned with relation to the order functions of PSTP since it considers only the remaining cost to reach the goal, not the total cost.)

The two features of GTM which vastly improve its performance relative to a more naive geometry theorem proving machine using only the theorems are: special handling of certain concepts (symmetry, recognition of identity of alternately named angles) and the use of the diagram to reject false subgoals.

2.14.2. Abstraction of Geometry-Theorem Proving Machine. Reiter [38,39] has tried to generalize the notion of models in fomal deductive systems. Using GTM as an example, he develops a formal system resembling Bledsoe et al.'s [5,8] IMPLY which prunes its search tree by evaluating subgoals in a model.

The extension of a formal system like Bledsoe's IMPLY to incorporate a model is quite straightforward. It is only

necessary to determine if some instance of generated subgoals is true in the model. If all instances are false, the subgoal is rejected.

A formal justification for the use of models to reject deductions containing false subgoals in the problem reduction method takes the form of the following proof of the compatibility of the use of models and input resolution. Input resolution is, of course, isomorphic to the problem reduction method, being transformed into it by the negation of literals in the deduction.

Theorem 2.9. (Henschen and Wos [20]) If S is a minimally unsatisfiable set of Horn clauses, then S contains exactly one negative clause.

Theorem 2.10. Let S be an unsatisfiable set of ground clauses. Let M be a model for the non-negative clauses of S expanded to the language of S. Then there exists an input refutation of S such that the negative top clause and every resolvent is false in M.

Proof. Proof by induction on the number of distinct atoms n in S.

n=1. If n=1 then S = {P, ¬P} for some atom P. The resolution of ¬P and P to derive the empty clause □ is an input refutation of S with negative top clause ¬P and single resolvent □ both false in M. Therefore, the theorem is true for this case.

n>1. Let P be (the atom of) a positive unit clause in S. There must exist such a clause since the lack of a positive

clause contradicts the unsatisfiability of S, and no positive Horn clause may have more than one literal by definition.

Let S_1 be S with (1) every clause containing P removed (if S is minimally inconsistent, only the clause P will be removed), and (2) every instance of $\neg P$ deleted. S_1 is an unsatisfiable Horn set with fewer than n distinct atoms. Also, M is a model for the non-negative clauses of S_1. Therefore, by the induction hypothesis, there exists an input refutation of S_1 such that the negative top clause and every resolvent is false in M.

Let D be the input refutation of S_1 such that the negative top clause and every resolvent is false in M. Add $\neg P$ to the top clause if the clause in S from which it was derived contained $\neg P$. Add $\neg P$ to each resolvent if the parent in S_1 was obtained from a clause in S containing $\neg P$, or if its near parent has had $\neg P$ added. If the resulting derivation has $\neg P$ as the result rather than the empty clause $\square$ as before, resolve it away with P to obtain $\square$. The result is an input refutation of S with the negative top clause and every resolvent false in M. $\square$

This is a vitally important instance of the more general theorem by Henschen [18] relating the use of models to deduction with Horn sets. His theorem does not restrict the model to falsifying the negative top clause of the deduction.

Reiter brings up some important issues relating to the use of models in first order predicate calculus theorem proving systems. In Reiter's system (and nearly all others that attempt to work with the first order predicate calculus) only quantifier free

formulas are employed, Skolem functions being previously introduced. Thus, for example, in a formulation of a GTM problem in Reiter's system, Skolem constants would be introduced to designate points in the premises and conclusion. The interpretation of the resulting formula is that it is to be proved true for every interpretation of the Skolem constants. The difficulty is that while the formula is to be proved true for every interpretation of the Skolem constants, the standard notion of a model allows a constant to represent only one element of the domain. An approach to solving this problem is, instead of employing a single model to test for rejection of subgoals, several "parallel models" can be employed, each representing the assignment of Skolem constants to points in the diagram which satisfies the premises This is really not an important issue in the geometry theorem proving example since there is likely to be only one non-redundant interpretation of Skolem constants as points in the diagram which satisfies the premises. An example which illustrates the utility of the parallel models approach is the group theory problem of proving that a group in which the square of every element is the identity element is Abelian (the GROUP2 problem in Appendix A). Here, our model can be an example of a group in which the square of each element is the identity element. The restriction that Skolem constants a and b from the statement of the theorem a*b=b*a be associated with single elements of the domain creates a rather weak model. In this example, we can generate several models each representing a

different interpretation of a and b as elements of the group and examine each subgoal in all these models.

Similar parallel models can be set up to handle different interpretations of non 0-ary Skolem functions. Suppose f is a unary Skolem function. Then to decide whether to reject subgoal Pxf(x), is is necessary to determine if for every parallel model there exists an assignment to x for which Pxf(x) is true. If not, the subgoal is rejected. Unfortunately, this approach yields a weaker model than necessary. Performing the above check is equivalent to check $\forall y \exists x Pxy$ whereas the expression $\exists x \forall y Pxy$ is a valid expression to check (since this is the formula from which Pxf(x) is derived by Skolemization). This gambit of testing the model satisfiability of an expression by analyzing the intent of the Skolem function is not always successful. The formula Pf(x)f(y) (obtainable by resolution for example) is an example of a formula uninterpretable in this way since it is not the product of Skolemizing any formula (where f is again assumed to be a Skolem function). This difficulty with formally describing the strongest possible model to use in the deductive system remains unresolved.

These previous ideas can readily be extended to infinite models by reducing the latter to finite models by choosing a finite set of representatives of the infinite model. For example, in number theory theorem proving, a model can be created by coding arithmetic operations +,-,*, etc. and supplying a small, finite subset of the natural numbers (e.g., 0, 1, 2, 5, 17, 24).

2.14.3. Types of models. Models can be evaluated in terms of two criteria: accuracy and scope.

The issue of accuracy of a model is best considered in the case of a complete theory and is concerned with whether the model ever falsely assigns a value to an expression.

Two entirely different effects may occur if the model being used is inaccurate. The first possibility is that the model will assign value true (i.e., provable, proceed) to a goal which is false. The second possibility is that the model will assign value false (i.e., disprovable, abort this proof) to a goal which is true. These two possibilities are similar to type I and type II errors in statistical inference respectively, where a type I error accepts a false non-null hypothesis and a type II error rejects a true non-null hypothesis. We will call these two possibilities type I and type II errors following the statistics convention.

In some circumstances, the commission of a type I error, though by retaining and attempting to prove a false goal it wastes computational effort, is harmless from the standpoint of completeness. This is the case in GTM. Here, if the model assigns value true to expressions which are false (by for example having in the diagram two angles which are equal within the tolerances of measurement which are not necessarily equal given the statement of the problem), only computational effort is wasted in possibly attempting to prove this false equality and completeness is not sacrificed.

The commission of a type II error is generally more serious since the response to the evaluation of an expression in the model to false results in the elimination of a partial deduction rather than the retention of one. It is possible in these circumstances to eliminate all valid proofs using a faulty model. Type II errors are avoided in GTM by correct construction of the diagram and using tolerances in measurement in determining what is true in the diagram to avoid false inference on the basis of measurement error.

In statistical inference, there is a symmetry between type I and type II errors and the null and non-null hypotheses. Committing a type I error is equivalent to committing a type II error with respect to the null hypothesis by rejecting it. Committing a type II error is equivalent to committing a type I error with respect to the null hypothesis by accepting it. This symmetry persists in the analogous case of models. If the model incorrectly evaluates an expression in the theory, either a type I or type II error will be committed when considering that expression as a goal and a type II or type I error respectively will be committed when the negation of the expression is considered as a goal.

Given this symmetry, and the fact that type II errors are generally considered to be intolerable since they may cut off the only path to a proof, how is it that GTM can tolerate even type I errors? The answer is that GTM never attempts to prove the negation of a goal. Thus, a type I error for a goal will never

result in a type II error for its negation because its negation will never occur.

Given this symmetry indicating that a type I error for an expression results in a type II error for its negation and vice versa, and the intolerability of type II errors, we must be concerned only with accurate models which incorrectly evaluate no expressions if we are concerned with a procedure which will be asked to prove expressions and their negations.

The scope of a model refers to the extent to which it defines values for expressions which actually are provable in the theory. In a theory, there are three possibilities for any expression: the expression is provable, the expression is disprovable (the negation of the expression is provable), and neither the expression nor the negation of the expression is provable. In the first case, the expression will be termed to be true; in the second case, the expression will be termed to be false; in the third case, the expression will be termed to be contingent since its value is contingent on information outside the theory.

A model which assigns values to all the non-contingent and some of the contingent expressions of the theory will be called an example since it is a model for an instance of the object described in the theory.

A model which assigns values to only some of the non-contingent and none of the contingent expressions of the theory will be called an abstraction since it is a model for a subtheory of the object described in the theory.

A model may fit neither category, assigning values to some contingent expressions and failing to assign values to some non-contingent expressions. Such a model will be called mixed.

In our future discussion, we will allow any of these types of models (examples, abstractions, mixed models). However, since we are concerned with problem solving procedures which deal with both goals and their negations, we must restrict ourselves, following the reasoning above, to accurate models to eliminate any possibility of eliminating all solutions by false inference from models.

The one sort of model we need not consider is that which assigns correct values to all non-contingent expressions and no values to contingent expressions. This model assigns values true or false to precisely those expressions which are provable or disprovable respectively. Since this is so, the model has exactly the same deductive power as the problem solving system, and there would be no need for the latter.

2.14.4. Incompatibility of π procedure with use of models. We could hypothesize that the π procedure is compatible with the use of models--that any deduction containing a false goal could be eliminated. It is readily apparent, however, that even refutable goals must sometimes be considered in the π procedure. The simplest example of proof by contradiction requiring the reduction operation (matching a goal to the negation of a higher goal) demonstrates this.

Example 2.25.

1. $\neg A \vee C$

2. A ∨ C	
3. C	theorem
4. [C] A	extend 3 by 1(a)
5. [C] [A] ¬C	extend 4 by 2(b)
6. ∅	reduce 5(a,c)

Note that the proof requires the presence of the goal ¬C, a refutable goal (indeed, the negation of the goal which we set out to prove). The alternative proof which extends first by clause 2, then by clause 1 resulting in intermediate goal ¬A rather than A suffers equally from this difficulty.

This proof precludes the possibility of a model refinement of the τ procedure which eliminates all proofs containing a refutable literal. We must then refine our definition of model unacceptable chains to allow chains whose literals which are false in the model are to be removed by reduction.

We can then hypothesize the completeness of use of models with the τ procedure if rejection of chains is eliminated where the cause of rejection is the falsity in the model of subgoals to be reduced. The following example with both possible proofs illustrates the incompleteness of this procedure.

Example 2.26.

1. ¬A ∨ C	
2. ¬B ∨ C	
3. A ∨ B	
4. C	theorem
5.1. [C] A	extend 4 by 1(b)
6.1. [C] [A] ¬B	extend 5.1 by 3(a)
7.1. [C] [A] [¬B] ¬C	extend 6.1 by 2(a)
8.1. ∅	reduce 7.1(a,d)
5.2. [C] B	extend 4 by 2(b)
6.2. [C] [B] ¬A	extend 5.2 by 3(b)
7.2. [C] [B] [¬A] ¬C	extend 6.2 by 1(a)
8.2. ∅	reduce 7.2(a,d)

The set of clauses {¬A ∨ C, ¬B ∨ C, A ∨ B} has 3 models: {A,B,C}, {¬A,B,C}, and {A,¬B,C}.

The first of the two proofs of C from these clauses contains no goals not in the model except for ¬C removed by reduction for the third model only. The second of the two proofs of C from these clauses contains no goals not in the model except for ¬C removed by reduction for the second model only. Note that neither proof is acceptable to the first model, and thus the * procedure is incompatible with the model refinement for arbitrary choice of models.

We can even modify the last example to illustrate that the * procedure must sometimes even expand subgoals which are disprovable (false in any example model).

Example 2.27.

1. ¬A ∨ C
2. ¬B ∨ C
3. A ∨ B
4. ¬C ∨ A
5. ¬C ∨ B

From the first clause we have that if A is true then C is true. From the second clause we have that if B is true then C is true. The third clause states that A or B is true; therefore, C must be true (by the first and second clauses). By the third clause we have that if C is true then A is true. By the fourth clause we have that if C is true then B is true. Since C is true by the reasoning above, both A and B must also be true. Thus, A, B, and C are all provable from this set of clauses and the only complete model is {A,B,C}.

The proposed model refinement states that partial deductions containing refutable goals not removable due to their being the negation of a higher goal are eliminable. Thus, the model refinement in this instance would permit the elimination of any partial deduction containing ¬A, ¬B, or ¬C except as the negation of a higher goal.

Following is a complete set of valid proofs of the C from the set of axioms.

6. C	theorem
7. [C] A	extend 6 by 1(b)
8. [C] [A] ¬B	extend 7 by 3(a)
9. [C] [A] [¬B] ¬C	extend 8 by 2(a)
10. ∅	reduce 9(a,d)

6. C	theorem
7. [C] B	extend 6 by 2(b)
8. [C] [B] ¬A	extend 7 by 3(b)
9. [C] [B] [¬A] ¬C	extend 8 by 1(a)
10. ∅	reduce 9(a,d)

The appearance of ¬C in each of these proofs is acceptable to the model refinement since in each case it is removed by reduction and is thus a negated higher goal. However, neither proof is totally acceptable to the model refinement if A, B, and C are all in the model (as they validly may be since all are provable) since one proof contains ¬A and the other contains ¬B.

2.14.5. A model refinement for the * procedure. If the * procedure restriction on eliminating partial deductions in which a goal is identical to a higher goal, and thus permitting a goal to be a subgoal of itself, is eliminated, the model refinement does accept a proof to the last example.

Example 2.28.

1. ¬A ∨ C
2. ¬B ∨ C

3.	$A \vee B$	
4.	$\neg C \vee A$	
5.	$\neg C \vee B$	
6.	C	theorem
7.	[C] A	extend 6 by 1(b)
8.	[C] [A] C	extend 7 by 4(b)
9.	[C] [A] [C] B	extend 8 by 2(b)
10.	[C] [A] [C] [B] $\neg A$	extend 9 by 3(b)
11.	$\emptyset$	reduce 10(b,e)

There is, by the symmetry of the set of axioms, a similar proof in which B rather than A is the first subgoal of C.

However, even this refinement does not permit a deduction to the earlier example of proving C from $\{\neg A \vee C, \neg B \vee C, A \vee B\}$ with the example model {A,B,C}. If we further extend the idea of allowing repetition of goals, even when the repetition cannot be formed by use of the axioms, a deduction for this example can be found.

Example 2.29.

1.	$\neg A \vee C$	
2.	$\neg B \vee C$	
3.	$A \vee B$	
4.	C	theorem
5.	[C] A	extend 4 by 1(b)
6.	[C] [A] C	repeat higher goal C
7.	[C] [A] [C] B	extend 6 by 2(b)
8.	[C] [A] [C] [B] $\neg A$	extend 7 by 3(b)
9.	$\emptyset$	reduce 8(b,e)

This leads to our final hypothesis of a complete combination of use of models and the π procedure: any chain can be eliminated if any A-literal (literal extended upon) is false in the model, provided identical A-literals may be duplicated. We will call this procedure the Mπ procedure for π procedure with model.

We can express the Mπ procedure in terms of a single inference

rule like the π procedure except that all higher subgoals are free to be extended upon, not just the lowest goal. Thus, we can consider trying to prove at each stage the disjunction of the lowest goal and all its higher goals. Following is a definition of the $M\pi$ procedure in terms of its single inference operation.

Definition 2.37. $B_1,\ldots,B_p \vdash_{M\pi} C_1 \vee \ldots \vee C_q$ if

B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C_j$ $(m \geq 0)$

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_1 \vdash_{M\pi} A_1 \vee C_1 \vee \ldots \vee C_q$

.:

and $B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_i \vdash_{M\pi} A_i \vee C_1 \vee \ldots \vee C_q$

.:

and $B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_m \vdash_{M\pi} A_m \vee C_1 \vee \ldots \vee C_q$ for some k,j $(1 \leq k \leq p, 1 \leq j \leq q)$.

If $m \neq 0$ then C_j must not be false in the model.

Theorem 2.11. The $M\pi$ procedure is sound, i.e., if $B_1,\ldots,B_p \vdash_{M\pi} C_1 \vee \ldots \vee C_q$ then $B_1,\ldots,B_p \vdash C_1 \vee \ldots \vee C_q$ where $\vdash_{M\pi}$ denotes derivability by the $M\pi$ procedure.

Proof. We will perform induction on the number of applications n of the single $M\pi$ inference rule used in a $M\pi$ procedure proof.

n=1. Assume $B_1,\ldots,B_p \vdash_{M\pi} C_1 \vee \ldots \vee C_q$ with only one application of the inference rule used in the proof. Since only one application of the inference rule is required, B_k must be some C_j $(1 \leq j \leq q)$. Otherwise, additional applications would be required to prove A_1, ..., and A_m. Clearly $B_1,\ldots,B_{k-1},C_j,B_{k+1},B_p \vdash C_1 \vee \ldots \vee C_q$. Therefore, the $M\pi$ procedure is sound for this case.

n>1. Assume the M_π procedure is sound for all proofs with fewer than n applications of the inference rule. Prove it is sound for n applications. Assume $B_1,\ldots,B_p \vdash_{M_\pi} C_1 \vee \ldots \vee C_q$ using n applications of the inference rule.

The proof of $C_1,\ldots,C_q$ employs the subproofs

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_1 \vdash_{M_\pi} A_1 \vee C_1 \vee \ldots \vee C_q$,

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_i \vdash_{M_\pi} A_i \vee C_1 \vee \ldots \vee C_q$, ..., and

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_m \vdash_{M_\pi} A_m \vee C_1 \vee \ldots \vee C_q$

where B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C_j$ for some k,j ($1 \leq k \leq p, 1 \leq j \leq q$). By the induction hypothesis, since each of these subproofs has fewer than n applications of the inference rule,

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_1 \vdash A_1 \vee C_1 \vee \ldots \vee C_q$,

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_i \vdash A_i \vee C_1 \vee \ldots \vee C_q$, ..., and

$B_1,\ldots,B_p,\sim C_1,\ldots,\sim C_q,\sim A_m \vdash A_m \vee C_1 \vee \ldots \vee C_q$.

Thus, by the deduction theorem,

$B_1,\ldots,B_p, \vdash A_1 \vee C_1 \vee \ldots \vee C_q$,

$B_1,\ldots,B_p, \vdash A_i \vee C_1 \vee \ldots \vee C_q$, ..., and

$B_1,\ldots,B_p, \vdash A_m \vee C_1 \vee \ldots \vee C_q$.

Therefore, $B_1,\ldots,B_p, \vdash (A_1 \wedge \ldots \wedge A_m) \vee C_1 \vee \ldots \vee C_q$. But since B_k is $A_1 \wedge \ldots \wedge A_m \rightarrow C_j$ we also have $B_1,\ldots,B_p \vdash \neg(A_1 \wedge \ldots \wedge A_m) \vee C_j$. Therefore, $B_1,\ldots,B_p \vdash [\neg(A_1 \wedge \ldots \wedge A_m) \vee C_j] \wedge [(A_1 \wedge \ldots \wedge A_m) \vee C_1 \vee \ldots \vee C_q]$ and, by simplification, $B_1,\ldots,B_p \vdash C_1 \vee \ldots \vee C_q$. Therefore, tne M_π procedure is sound for this case. □

We do not yet have a proof for the completeness of the M_π procedure, but know no counter-examples to it.

There is a hierarchy of procedures which can be applied when using models, each capable of proving more theorems than its predecessor in the hierarchy: the unmodified π procedure with identical subgoal elimination rule, the π procedure without the identical subgoal elimination rule, and the Mπ procedure with totally free repetition of goals.

The incompatibility of the higher identical subgoal elimination rule and the use of models forces a tradeoff between the use of models and the single most useful elimination rule. Thus, if the model being used is weak or non-existent, it may be more fruitful to eliminate its use and retain the higher identical elimination rule.

However, the loss of the higer subgoal elimination rule is not total since weaker forms of the higher identical subgoal elimination rule may be substituted while retaining the completeness of the use of models.

Consider again the last example which demonstrated the incompatibility of the unmodified π procedure with the use of models.

Example 2.30.

1. $\neg A \vee C$
2. $\neg B \vee C$
3. $A \vee B$
4. $\neg C \vee A$
5. $\neg C \vee B$

Possible derivations from the top goal C include:

6. C	theorem
7.1. [C] A	extend 6 by 1
7.2. [C] B	extend 6 by 2

8.1.1. [C] [A] ¬B	extend 7.1 by 3
8.1.2. [C] [A] C	extend 7.1 by 4
8.2.1. [C] [B] ¬A	extend 7.2 by 3
8.2.2. [C] [B] C	extend 7.2 by 5
9.1.1. [C] [A] [¬B] ¬C	extend 8.1.1 by 2
9.1.2.1. [C] [A] [C] A	extend 8.1.2 by 1
9.1.2.2. [C] [A] [C] B	extend 8.1.2 by 2
9.2.1. [C] [B] [¬A] ¬C	extend 8.2.1 by 1
9.2.2.1. [C] [B] [C] A	extend 8.2.2 by 1
9.2.2.2. [C] [B] [C] B	extend 8.2.2 by 2
10.1.1. ∅	reduce 9.1.1
10.1.2.1.1. [C] [A] [C] [A] ¬B	extend 9.1.2.1 by 3
10.1.2.1.2. [C] [A] [C] [A] C	extend 9.1.2.1 by 4
10.1.2.2.1. [C] [A] [C] [B] ¬A	extend 9.1.2.2 by 3
10.1.2.2.2. [C] [A] [C] [B] C	extend 9.1.2.2 by 4
10.2.1. ∅	reduce 9.2.1
10.2.2.1.1. [C] [B] [C] [A] ¬B	extend 9.2.2.1 by 3
10.2.2.1.2. [C] [B] [C] [A] C	extend 9.2.2.1 by 4
10.2.2.2.1. [C] [B] [C] [B] ¬A	extend 9.2.2.2 by 3
10.2.2.2.2. [C] [B] [C] [B] C	extend 9.2.2.2 by 5

Consider chains 9.1.2.1 ([C] [A] [C] A) and 9.2.2.2 ([C] [B] [C] B). Both these chains can be eliminated with no loss of completeness. The justification for this is that every operation or sequence of operations employed to remove the terminal literal A of chain 9.1.2.1 or the terminal literal B of chain 9.2.2.2 can equally well be employed to remove the earlier A or B. The critical observation is that in each of these chains the new terminal literal has the same set of nigher non-identical subgoals as an earlier occurrence of that same literal, i.e., A has only higher non-identical subgoal C in both occurrences in chain 9.1.2.1 and B has only higher non-identical subgoal C in both occurrences in chain 9.2.2.2.

The retention of needed chains such as 8.1.2 ([C] [A] C) and 8.2.2 ([C] [B] C) which also have higher identical goals is

justified since the new later occurrences of C appears with a higher goal A (in 8.1.2) or B (in 8.2.2) which was not a higher goal of the earlier occurrences of C.

The rule can be expressed as: any deduction in which a goal has the same set of higher non-identical (to itself) goals as a higher occurrence of the same goal can be eliminated.

3. Unification Algorithms for Artificial Intelligence Languages

3.1. Introduction

An extremely important component of many artificial intelligence languages is a general pattern matcher for symbolic expressions. In present artificial intelligence languages, pattern matching is typically uni-directional, i.e., the expressions are not treated symmetrically permitting variable assignments in both expressions. (Indeed, we will often use the term pattern matching to refer to the case of unification in which at least one of the two expressions has no variables.) This is a major source of incompleteness since it is clear that some form of unification (which is bi-directional) is required for completeness. In addition to the incompleteness aspect of this uni-directional pattern matching, even where completeness and full capability for unification is not required, it is often desirable to have at least a limited bi-directional pattern matching capability. For example, in QLISP it would be desirable if when (GOAL (P ←X ↔Y) ...) is attempted, functions with pattern (P 0 ↔Z) would be evoked as well as patterns with greater or equal generality than the evoking pattern (P ←X ↔Y).

The solution which we argue for here is based on two components: semi-unification and variable splitting (or, in more general terms, semi-unification and elaboration).

Semi-unification algorithms are presented for important data types: vectors, bags, and classes. These algorithms return the most immediate and natural (in general incomplete) set of unifiers for two expressions. Semi-unification can be used without variable splitting for languages and applications where completeness is not required but some bi-directional pattern matching capability is. Where completeness is required, the additional process of variable splitting or elaboration can be included.

In order to motivate this solution, let us consider the alternatives.

The first possibility is, of course, to merely write complete unification algorithms for the desired data types and be done with it. There are theoretical difficulties in this approach. One of the data types treated here, the vector, is simply a sequence of elements in order with possible repetition of the elements. The use of fragment variables, which in pattern matching permit the matching of the variable to an arbitrary number of elements in the opposing expression, results in the possible generation of an infinite number of unifiers for vector unification. A typical example of this phenomenon is the unification of expressions (VECTOR A ↔X) and (VECTOR ↔X A) where A is a constant and X is a fragment variable. Since X can match an arbitrary number of elements, the two expressions have unifiers with X assigned the empty vector, the vector consisting of one A, the vector consisting of two A's, etc, with unifying

expressions (VECTOR A), (VECTOR A A), (VECTOR A A A), etc., respectively.

This problem does not occur for the other two data types considered here. We present complete unification algorithms for bags and classes. It may also be possible to write a complete unification algorithm for vectors which adopts a more general representation of unifiers which finitely expresses an infinite set of unifiers. For example, the entire set of unifiers of the two expressions above could be expressed as $X \leftarrow (\text{VECTOR } A^n)$ for all $n \geq 0$.

Even where this approach is possible (as for bags and classes), it is not always appropriate. The striving for logical completeness in pattern matching obviously has costs, and language designers and users are not always willing to pay those costs. Except in those cases where completeness is required in essentially all usage of the system, an approach which permits completeness while also allowing for more restrictive incomplete pattern matching is appropriate.

A second alternative is the deferral of completion of pattern matching until the skeleton of a solution is found. The solution is correct provided the deferred matches succeed. This is the approach of Huet's constrained resolution [58]. It has the advantage of not requiring complete unification at each step in a deduction, but allowing for the demonstration of the existence of a set of unifiers at the end of a trial deduction. This substitution of determining the existence of unifiers for sets of

expressions for the computing of complete sets of unifiers (i.e., demonstrating existence of unifiers rather than computing most general unifiers) can result in a substantial saving of computational effort as well as bypassing theoretical issues concerning computation of most general unifiers. This approach has the disadvantage in the design of Artificial Intelligence Languages of employing not fully instantiated goals and subgoals (not fully instantiated because the pattern matching is being deferred). This lack of instantiation will inhibit the full power of the heuristics programmed in the language to select among goals to be attempted. With respect to the programmed heuristics, it is more advantageous to have fully instantiated goals which can be evaluated for rejection or interest than to save some computation by leaving goals uninstantiated.

A third alternative is the use of a parameterized incomplete pattern matching or unification algorithm. Such an algorithm would, for example, return unifiers of increasing complexity for increasing values of a numerical parameter. A problem solver using such an algorithm could, upon failure, automatically backtrack to a previous pattern matching or unification operation, increase the parameter value, and continue with an expanded set of unifiers. The principal disadvantage of this approach is the lack of control of the system user over the unifiers being returned. The mechanism for generating additional unifiers is hidden from him.

The fourth approach, the one being adopted here, provides the

user with this type of control. The operations which, when applied to the expressions being unified, result in the generation of additional unifiers are explicitly available to him. The generality of this approach as compared to the third approach is that this approach could be used to implement the second approach.

3.2. Terminology

Expressions are written in inverse_quote_mode. In contrast to LISP in which expressions are evaluated unless explicitly quoted, in inverse quote mode expressions are evaluated only if preceded by a prefix character or characters indicating a variable to be evaluated or assigned a value. The function of elements of expressions (whether the element is a constant or variable) is denoted by the absence or presence and type of a prefix as opposed to alphabetical conventions as in formal logic.

Definition 3.1. An expression is defined to be

(1) a constant,

(2) a simple variable, or

(3) a list consisting of an expression (the head of the expression) followed by an arbitrary number of elements (the tail of the expression).

Expressions which are constants or simple variables are termed atomic expressions; other expressions are termed non-atomic expressions.

Definition 3.2. An element (of an expression) is defined to be

(1) an expression, or

(2) a fragment variable.

An element will be represented by lower case Latin letters a, b, etc. A sequence of elements will be represented by lower case Greek letters α, β, etc.

Definition 3.3. A ground_expression is an expression with no variables.

Definition 3.4. The type of each non-atomic expression is determined by its head. The expression type designates the storage and pattern matching or unification discipline to be used for the expression. Only expressions with the same type can be successfully matched. Three types are presently defined: vectors, bags, and classes.

Definition 3.5. A vector is an ordered sequence of elements permitting repetition of elements. It can represent the argument list of an associative or non-associative function with a fixed or variable number of arguments.

Definition 3.6. A bag is an unordered sequence of elements permitting repetition of elements. It can represent the argument list of an associative or non-associative commutative function with a fixed or variable number of arguments.

Definition 3.7. A class is an unordered sequence of elements

not permitting repetition of elements. It is a mathematical set and can represent the argument list of an associative or non-associative function with set argument.

Expression heads VECTOR, BAG, and CLASS have type vector, bag, and class respectively.

Examples of functions with vector arguments are the LISP functions "list" and "append". The "list" function takes an arbitrary number of arguments, but is not associative, i.e., list[x;list[y;z]]≠list[list[x;y];z]. The "append" function takes an arbitrary number of arguments and is associative.

Examples of functions with bag arguments are the arithmetic functions "plus" and "mean". The "plus" function takes an arbitrary number of arguments and is associative. The "mean" function takes an arbitrary number of arguments and is not associative.

Examples of functions with class arguments are the logical functions "and" and "equivalence". Both are associative.

Definition 3.8. A constant is defined to be
(1) a numerical constant, e.g., 3, -2, 4.5, or
(2) a symbolic constant, e.g., 12A, X, ABC, but not 12, +X. A symbolic constant is represented as a string of characters not interpretable as a numerical constant or a variable.

Definition 3.9. A variable is represented as a symbolic constant preceded by a prefix. Prefixes are +, ++, +++, $, $$, and $$$.

Definition 3.10. A simple variable is a variable whose prefix is one of + and $.

Definition 3.11. A fragment variable is a variable whose prefix is one of ++, +++, $$, and $$$.

The different variable prefixes have different interpretations in the context of a pattern matching or unification operation. The three prefixes using + denote assignment of values to variables regardless of past variable values. The three prefixes using $ require past variable values to be matched in the matching expression. If the variable has not previously been assigned a value, $ prefixes behave like their + counterparts, i.e., $X acts as +X, $$X acts as ++X, and $$$X acts as +++X.

+ prefixes. Three kinds of + prefixes are used:

(1) + represents the matching of the variable to a single expression,

(2) ++ represents the matching of the variable to zero or more elements,

(3) +++ represents the matching of the variable to one or more elements.

In a pattern matching or unification operation, the value assigned to the variable is an expression computed from the matching expression. In the case of simple variables (with prefix + or $), the value assigned is simply the matched expression. In the case of fragment variables (with prefix ++, +++, $$, or $$$), the value assigned is an expression whose head is the same as the head of the expression whose elements are being matched and whose tail is the sequence of matched elements.

$ prefixes. Three kinds of $ prefixes are used:

(1) in any context, the evaluation of $X results in the value of X,

(2) in the context of an expression whose type is the same as the type of the value of X, the evaluation of $$X results in the insertion of the elements of the value of X into the expression; in other contexts, the evaluation of $$X is illegal,

(3) in the context of an expression whose head is the same as the head of the value of X, the evaluation of $$$X results in the insertion of the elements of the value of X into the expression; in other contexts, the evaluation of $$$X results in the value of X if it has other than one element, the single element of its value if it has exactly one element.

Example 3.1. Assume F and G are expression heads of type vector and H is an expression head of type bag. The first column gives values of variable X; the remaining columns give the result of evaluating (substituting into) various expressions.

X	$X	$$X	$$$X
A	A	illegal[1]	illegal[1]
(F)	(F)	(VECTOR)[2]	(F)[2,3]
(F B)	(F B)	(VECTOR B)[2]	B[2]
(F C D)	(F C D)	(VECTOR C D)[2]	(F C D)[2]

X	(F I $X J)	(F I $$X J)	(F I $$$X J)
A	(F I A J)	illegal[1]	illegal[1]
(F)	(F I (F) J)	(F I J)	(F I J)[3]
(F B)	(F I (F B) J)	(F I B J)	(F I B J)
(F C D)	(F I (F C D) J)	(F I C D J)	(F I C D J)

X	(G I $X J)	(G I $$X J)	(G I $$$X J)
A	(G I A J)	illegal[1]	illegal[1]
(F)	(G I (F) J)	(G I J)	(G I (F) J)[3]
(F B)	(G I (F B) J)	(G I B J)	(G I B J)
(F C D)	(G I (F C D) J)	(G I C D J)	(G I (F C D) J)

X	(H I $X J)	(H I $$X J)	(H I $$$X J)
A	(H I A J)	illegal[1]	illegal[1]
(F)	(H I (F) J)	illegal[4]	(H I (F) J)[3]
(F B)	(H I (F B) J)	illegal[4]	(H I B J)
(F C D)	(H I (F C D) J)	illegal[4]	(H I (F C D) J)

[1]X must have a non-atomic expression as its value for $$X or$$$X to be successfully evaluated.

[2]This represents an extended definition of expressions. The original definition stated that $$X and $$$X are elements but not expressions.

[3]This case should rarely if ever occur. It is intended that ... prefix variables be matched with non-empty expression sequences.

[4]The type of the expression containing the $$X must agree with the type of the value of X.

Note that these rules for variable prefixes differ slightly from those of the latest version of QLISP [44,53]. In the latest QLISP, the value of a fragment variable will be "stripped" when instantiated in some other expression, i.e., its outer parentheses and type will be removed, regardless of whether the type of the fragment variable and the type of the expression agree. For example, if X has the value (VECTOR C D), (BAG I $$X J) would have the value (BAG I C D J). This is a pragmatically reasonable interpretation of $$X, but is less reasonable logically.

Definition 3.12. A substitution component is an ordered pair of a variable v and an expression e written as v←e. A substitution component denotes the assignment of the expression to the variable or the replacement of the variable by the expression.

Definition 3.13. A substitution is a set of substitution components with distinct first elements, i.e., distinct variables being substituted for. Applying a substitution to an expression results in the replacement of those variables of the expression included among the first elements of the substitution components by the corresponding expressions. The precise form of replacement depends on the prefix of the variable, i.e., whether it is a simple or fragment variable, as discussed above. The substitution components are applied to the expression in parallel and no variable occurring in the second element of a substitution

component will be replaced even if it occurs as the first element in another substitution component. Substitutions will be represented by lower case Greek letters σ, θ. The application of substitution θ to expression x is denoted by $x\theta$. The composition of substitutions $\theta\sigma$ denotes the substitution whose effect is the same as first applying substitution θ, then applying substitution σ, i.e., $x(\theta\sigma) = (x\theta)\sigma$ for every expression x.

Definition 3.14. A unifying substitution or unifier of two expressions is a substitution which when applied to the two expressions results in equivalent expressions. Two vectors are equivalent if they are identical; two bags are equivalent if they contain the same elements in the same number; two classes are equivalent if they contain the same elements. (In an actual implementation, bags and classes would ordinarily be retained only in a canonical form so that equivalent bags and classes would have identical canonical forms. Bags and classes can be put into a canonical form by applying a sorting function on the elements to order them in a standard way and eliminating duplicate elements of classes. In the following, we will often overlook the distinction between identical and merely equivalent expressions.)

Definition 3.15. One expression is an instance of another if there exists a substitution whose application to the second expression results in the first.

Definition 3.16. One expression is a ground instance of

another if it is a ground expression and is an instance of the other expression.

Definition 3.17. One expression is a generalization of another if the second is an instance of the first.

Similarly, one substitution is an instance (generalization) of another if for every expression the result of applying the first substitution to the expression is an instance (generalization) of the result of applying the second substitution to the expression.

3.3. Semi-unification and variable splitting

One way of providing completeness in the pattern matching process in artificial intelligence languages is to separate the pattern matching process into two components.

One of these components is the current pattern matching process or some limited extension of it. The intent here is to provide a pattern matching procedure which returns a subset of the unifiers of two expressions. This subset of the unifiers contains the most natural and necessary unifiers--those that even language designers not concerned with completeness should expect to be returned by their pattern matcher. This first component would return this subset of the unifiers with little more computational effort than conventional pattern matchers thereby eliminating objections that providing completeness is too costly.

The second component is a process which can be performed on a

pair of expressions to cause the first component to compute additional unifiers. Every unifier should be (an instance of) a unifier generated by the first component after some number of applications of the second component to the expressions.

There are numerous possible ways of designing these two components. A large class of these are described in the section "Semi-unification and elaboration".

Our choice here is to let the first component be a slight extension of the ordinary pattern matching process to permit variables of each expression to be assigned values in the pattern matching process. It has already been argued that this sort of extension is useful even in the absence of total completeness, thereby justifying its inclusion in the first (incomplete) component rather than the (completeness providing) second component. This component is called semi-unification since it has the characteristics of unification in its ability to assign values to variables of both expressions but is incomplete and cannot therefore be simply called unification.

Semi-unification as presented here is complete in the case where no fragment variables are present. Its incompleteness in the case where fragment variables are present results from its failure to consider one fragment variable including in its value part of another. Our second component facilitates such assignments being made in semi-unification by replacing fragment variables by two new fragment variables (splitting the variable). This process is called variable splitting.

For example, consider the matching of the expressions (VECTOR ←←X A) and (VECTOR B ←←Y). Semi-unification is just essentially a bi-directional pattern matching process which returns a single unifier: {X←(VECTOR B), Y←(VECTOR A)}. This is incomplete because such unifiers as {X←(VECTOR B C), Y←(VECTOR C A)} are not discovered. The failure to discover (a generalization of) the latter unifier is a consequence of semi-unification not including parts of fragment variables in the value of another.

A single application of variable splitting, replacing either ←←X or ←←Y by two new fragment variables, results in the generation of the most general unifier of the two expressions by semi-unification applied to the expressions after variable splitting. Splitting ←←X, (VECTOR ←←X1 ←←X2 A) and (VECTOR B ←←Y) are obtained. Semi-unification returns unifiers including {X1←(VECTOR B), Y←(VECTOR ←←X2 A)} or, displaying the value of X, {X←(VECTOR B ←←X2), Y←(VECTOR ←←X2 A)}.

We will present in detail semi-unification algorithms for vectors, bags, and classes, prove that they terminate and are sound (return no false unifiers), and that semi-unification plus variable splitting is complete.

We will assume in the following that every non-atomic expression is either a vector, bag, or class, that values for bound variables have already been substituted in the expressions so that the only remaining variables are unbound, and that only fragment variables with the ←← prefixes are used. Extension to typed expression heads other than VECTOR, BAG, or CLASS and ←←← prefixes is straightforward.

3.4. Semi-unification

The procedures unify and semi-unify are described below in the one procedure (semi-)unify. The string "(semi-)unify" should be uniformly replaced in the text of the procedure by either "unify" or "semi-unify" to generate the two procedures. The unify procedure is intended to be the central routine of a complete procedure for unifying expressions (if such a thing exists). The semi-unify procedure is intended to be the central routine of an incomplete procedure for unifying expressions which, in concert with variable splitting, yields completeness.

The unify and semi-unify procedures themselves handle the cases where one or both of the expressions is a constant or variable; other cases are handled by calling the appropriate vector, bag, or class unification or semi-unification procedure.

The unify and semi-unify procedures, and each of the subordinate procedures, return a set of all the unifying substitutions found. If the expressions are not unifiable, the empty set is returned.

Examples of vector, bag, and class semi-unification appear at the end of the descriptions of the algorithms.

(semi-)unify[x; y]=

1. If x=y then return {{}}.
2. If x is a simple variable of the form ←X and
   a. y is a simple variable or constant then return {{X←y}}.
   b. y is a vector of the form (VECTOR β) and

1. X is not contained in β then return $\{\{X \leftarrow y\}\}$.

2. X is contained in β then let $\theta = \{X \leftarrow (\text{VECTOR } \leftrightarrow T)\}$ and return $\{\theta\sigma \mid \sigma \in (\text{semi-})\text{unify}[(\text{VECTOR } \leftrightarrow T); y\theta]\}$.

c. y is a bag of the form (BAG β) and

1. X is not contained in β then return $\{\{X \leftarrow y\}\}$.

2. X is contained in β then let $\theta = \{X \leftarrow (\text{BAG } \leftrightarrow T)\}$ and return $\{\theta\sigma \mid \sigma \in (\text{semi-})\text{unify}[(\text{BAG } \leftrightarrow T); y\theta]\}$.

d. y is a class of the form (CLASS β) and

1. X is not contained in β then return $\{\{X \leftarrow y\}\}$.

2. X is contained in β then let $\theta = \{X \leftarrow (\text{CLASS } \leftrightarrow T1 \leftrightarrow T2)\}$ and return $\{\theta\sigma \mid \sigma \in (\text{semi-})\text{unify}[(\text{CLASS } \leftrightarrow T1 \leftrightarrow T2); y\theta]\}$.

3. If y is a simple variable of the form $\leftarrow Y$ and

a. x is a constant then return $\{\{Y \leftarrow x\}\}$.

b. x is a vector of the form (VECTOR α) and

1. Y is not contained in α then return $\{\{Y \leftarrow x\}\}$.

2. Y is contained in α then let $\theta = \{Y \leftarrow (\text{VECTOR } \leftrightarrow T)\}$ and return $\{\theta\sigma \mid \sigma \in (\text{semi-})\text{unify}[x\theta; (\text{VECTOR } \leftrightarrow T)]\}$.

c. x is a bag of the form (BAG α) and

1. Y is not contained in α then return $\{\{Y \leftarrow x\}\}$.

2. Y is contained in α then let $\theta = \{Y \leftarrow (\text{BAG } \leftrightarrow T)\}$ and return $\{\theta\sigma \mid \sigma \in (\text{semi-})\text{unify}[x\theta; (\text{BAG } \leftrightarrow T)]\}$.

d. x is a class of the form (CLASS α) and

1. Y is not contained in α then return $\{\{Y \leftarrow x\}\}$.

2. Y is contained in α then let $\theta = \{Y \leftarrow (\text{CLASS } \leftrightarrow T1 \leftrightarrow T2)\}$ and return $\{\theta\sigma \mid \sigma \in (\text{semi-})\text{unify}[x\theta; (\text{CLASS } \leftrightarrow T1 \leftrightarrow T2)]\}$.

4. If x is a constant or y is a constant then return $\{\}$.

5. If x is a vector and

a. y is a vector then (semi-)unifyvector[x; y].

b. y is not a vector then return {}.

6. If x is a bag and

a. y is a bag then (semi-)unifybag[x; y].

b. y is not a bag then return {}.

7. If x is a class and

a. y is a class then (semi-)unifyclass[x; y].

b. y is not a class then return {}.

Notes:

1. If the two expressions being unified are equal, the null substitution is returned as the only unifier (step 1).

2. If one of the expressions is a simple variable, the substitution of the other expression for the variable is returned as the only unifier provided the variable is not contained in the other expression (steps 2a, 3a, 2b1, 2c1, 2d1, 3b1, 3c1, 3d1). If both expressions are variables, the second will be substituted for the first. If the variable is contained in the expression and the expression is a vector, bag, or class, a vector, bag, or class consisting of new fragment variables will be assigned to the variable and unification will proceed (steps 2b2, 2c2, 2d2, 3b2, 3c2, 3d2). This permits the discovery of unifiers for such unifications as ←X with (VECTOR ←←X) (X is an arbitrary vector) and ←X with (CLASS ←←X ←←Y) (X is an arbitrary class, Y is an arbitrary subclass of X).

3. Non-identical constants are not unifiable (step 4).

4. Only vectors can be unified with vectors, only bags can be

unified with bags, and only vectors can be unified with vectors (steps 5, 6, 7).

3.5. Vector semi-unification

The semi-unifyvector procedure returns an incomplete list of unifiers for two vectors.

The semi-unifyvector procedure, when presented with two non-empty vectors, always removes the first element of the first vector at each step.

When presented with two vectors whose first elements are not fragment variables, it directly unifies the first elements and then unifies the instantiated remainders of the vectors. When one or both of the first elements of the vectors are fragment variables, each possible substitution of an initial subvector (including the empty vector) from the opposing vector for the fragment variable is tried.

semi-unifyvector[x; y]=

1. If x is of the form (VECTOR) and
   a. y is of the form (VECTOR ↔Y β) then let θ={Y←(VECTOR)} and return {θσ | σ ∈ semi-unify[x; (VECTOR β)θ]}.
   b. y is not of the form (VECTOR ↔Y β) then return {}.
2. If y is of the form (VECTOR) and
   a. x is of the form (VECTOR ↔X α) then let θ={X←(VECTOR)} and return {θσ | σ ∈ semi-unify[(VECTOR α)θ; y]}.
   b. x is not of the form (VECTOR ↔X α) then return {}.

3. If x is of the form (VECTOR a α) and y is of the form (VECTOR a β) for arbitrary element a then semi-unify[(VECTOR α); (VECTOR β)].

4. If x is of the form (VECTOR ↔X α) then

   a. if y is of the form (VECTOR ↔Y β) then for each partition of x into (VECTOR α γ) for which Y is not contained in α let θ={Y←(VECTOR α)} and return {θσ | σ ∈ semi-unify[(VECTOR γ)θ; (VECTOR β)θ]} and

   b. for each partition of y into (VECTOR β δ) for which X is not contained in β let θ={X←(VECTOR β)} and return {θσ | σ ∈ semi-unify[(VECTOR α)θ; (VECTOR δ)θ]}.

5. If x is of the form (VECTOR a α) where element a is not a fragment variable and

   a. y is of the form (VECTOR ↔Y β) then for each partition of x into (VECTOR α γ) for which Y is not contained in α let θ={Y←(VECTOR α)} and return {θσ | σ ∈ semi-unify[(VECTOR γ)θ; (VECTOR β)θ]}.

   b. y is of the form (VECTOR b β) where element b is not a fragment variable then return {θσ | θ ∈ semi-unify[a; b] ∧ σ ∈ semi-unify[(VECTOR α)θ; (VECTOR β)θ]}.

Notes:

1. If one of the vectors is the empty vector and the other is not, the latter must be composed solely of fragment variables which must be bound to the empty vector. Otherwise, there is no unifier (steps 1, 2).

2. If the first elements of two vectors are identical, the remainders of the vectors are unified (step 3).

3. If the first element of one or both of the vectors is a fragment variable, each substitution of any initial subvector of the opposing vector not containing the variable is performed and the remainders of the vectors are unified for each substitution (steps 4a, 4b, 5a).

4. If neither of the first elements of the vectors is a fragment variable, the two first elements are unified against each other and the remainders of the vectors are unified for each unifier of the first elements (step 5b).

3.6. Bag semi-unification

The semi-unifybag procedure returns an incomplete list of unifiers for two bags.

The semi-unifybag procedure first eliminates by pairs any element appearing in both bags.

The semi-unifybag procedure, when presented with two non-empty bags with no common elements, always removes the first element of the first bag.

When presented with a first bag whose first element is not a fragment variable, each match between the element and each element which is not a fragment variable in the second bag is attempted. If the match is successful, the remainders of each of the bags are unified. When the first element of the first bag is a fragment variable, each possible substitution of a subbag (collection of elements from a bag) from the second bag is tried.

In either case, each possible substitution for any fragment variable in the second bag of any subbag of the first bag which contains the first element is also tried.

semi-unifybag[x; y]=

1. If x is of the form (BAG) and
   a. y is of the form (BAG ↔Y β) then let θ={Y←(BAG)} and return {θσ | σ ∈ semi-unify[x; (BAG β)θ]}.
   b. y is not of the form (BAG ↔Y β) then return {}.
2. If y is of the form (BAG) and
   a. x is of the form (BAG ↔X α) then let θ={X←(BAG)} and return {θσ | σ ∈ semi-unify[(BAG α)θ; y]}.
   b. x is not of the form (BAG ↔X α) then return {}.
3. If x is of the form (BAG α a γ) and y is of the form (BAG β a δ) for arbitrary element a then semi-unify[(BAG α γ); (BAG β δ)].
4. If x is of the form (BAG ↔X α) then
   a. for each fragment variable ↔Y in y (y possibly reordered is of the form (BAG ↔Y β)) and for each partition of x into (BAG α γ) for which Y is not contained in α which does include first element ↔X let θ={Y←(BAG α)} and return {θσ | σ ∈ semi-unify[(BAG γ)θ; (BAG β)θ]} and
   b. for each partition of y into (BAG β δ) for which X is not contained in β let θ={X←(BAG β)} and return {θσ | σ ∈ semi-unify[(BAG α)θ; (BAG δ)θ]}.
5. If x is of the form (BAG a α) where element a is not a fragment variable then

a. for each fragment variable ↔Y in y (y possibly reordered is of the form (BAG ↔Y β)) and for each partition of x into (BAG α γ) for which Y is not contained in α which does include first element a let θ={Y←(BAG α)} and return {θσ | σ ∈ semi-unify[(BAG γ)θ; (BAG β)θ]} and

b. for each element b in y which is not a fragment variable (y possibly reordered is of the form (BAG b β)) return {θσ | θ ∈ semi-unify[a; b] ∧ σ ∈ semi-unify[(BAG α)θ; (BAG β)θ]}.

Notes:

1. If one of the bags is the empty bag and the other is not, the latter must be composed solely of fragment variables which must be bound to the empty bag. Otherwise, there is no unifier (steps 1, 2).

2. If two elements of two bags are identical, the remainders of the bags are unified (step 3).

3. If the first element of x is a fragment variable or any element of y is a fragment variable, each substitution of any subbag of the opposing bag not containing the variable and substituting for or including in the substitution the first element of x is performed and the remainders of the bags unified for each substitution (steps 4a, 4b, 5a).

4. If the first element of x is not a fragment variable, it is unified with each element of y which is not a fragment variable and the remainders of the bags are unified for each unifier of the two elements (step 5b).

3.7. Class semi-unification

The semi-unifyclass procedure returns an incomplete list of unifiers for two classes.

Since class unification requires an element of one class possibly to be matched against more than one element of another class, matched elements cannot be simply removed and disregarded as in the vector and bag semi-unification algorithms. Previously matched elements must be retained. They may be kept in an additional variable in the algorithm or, for simplicity of description and implementation, be retained in the original classes themselves as in the algorithm being presented here. The following conventions are followed by the algorithm: previously matched elements of two classes are retained in the right end (tail) of the two classes and the leftmost of any two duplicate elements in a single class is immediately eliminated. Two classes are fully unified when they are identical (all the elements are previously matched elements in the common tail of the two classes).

The semi-unifyclass procedure, when presented with two non-empty classes with no previously unmatched common elements or duplicate elements within a single class, always matches the first element of the first class at each step after a possible interchange of the two classes. The interchange is a normalization operation whose result is that the first element of

the first class after the interchange is guaranteed to be previously unmatched.

Class semi-unification proceeds almost exactly as does bag semi-unification except for the retention of matched expressions at the right end of the classes and (thus) the ability to match a previously unmatched element against elements previously matched.

semi-unifyclass[x; y]=

1. If x is of the form (CLASS α a γ a ϵ) for arbitrary element a then semi-unify[(CLASS α γ a ϵ); y].
2. If y is of the form (CLASS β a δ a ϵ) for arbitrary element a then semi-unify[x; (CLASS β δ a ϵ)].
3. If x is of the form (CLASS α a γ) and y is of the form (CLASS β a δ) for arbitrary element a and $\gamma \neq \delta$ then semi-unify[(CLASS α γ a); (CLASS β δ a)].
4. If x is of the form (CLASS α) and y is of the form (CLASS β α) then semi-unify[y; x].
5. If x is of the form (CLASS $\leftrightarrow$X α) then
   a. for each fragment variable $\leftrightarrow$Y in y (y possibly reordered is of the form (CLASS $\leftrightarrow$Y β)) and for each partition of x into (CLASS α γ) for which Y is not contained in α which does include first element $\leftrightarrow$X let θ={Y$\leftarrow$(CLASS α)} and return {$\theta\sigma$ | $\sigma \in$ semi-unify[(CLASS γ α)θ; (CLASS β α)θ]} and
   b. for each partition of y into (CLASS β δ) for which X is not contained in β let θ={X$\leftarrow$(CLASS β)} and return {$\theta\sigma$ | $\sigma \in$ semi-unify[(CLASS α β)θ; (CLASS δ β)θ]}.

6. If x is of the form (CLASS a α) where element a is not a fragment variable then

   a. for each fragment variable ↔Y in y (y possibly reordered is of the form (CLASS ↔Y β)) and for each partition of x into (CLASS α γ) for which Y is not contained in α which does include first element a let θ={Y←(CLASS α)} and return {θσ | σ ∈ semi-unify[(CLASS γ α)θ; (CLASS β α)θ]} and

   b. for each element b in y which is not a fragment variable (y possibly reordered is of the form (CLASS b β)) return {θσ | θ ∈ semi-unify[a; b] ∧ σ ∈ semi-unify[(CLASS α a)θ; (CLASS β a)θ]}.

Notes:

1. If a class has a repeated element, the first occurrences of that element are deleted (steps 1, 2).

2. If two elements of two classes not in the common tail of the classes (not previously matched) are identical, the classes with the elements moved to the end are unified (step 3).

3. To insure that the next substitution operation use a previously unmatched element, the classes are interchanged if the second is a tail of the first (step 4).

4. If the first element of x is a fragment variable or any element of y is a fragment variable, each substitution of any subclass of the opposing class not containing the variable and substituting for or including in the substitution the first element of x is performed and the remainder of the classes are unified for each substitution (steps 5a, 5b, 6a).

5. If the first element of x is not a fragment variable, it is unified with each element of y which is not a fragment variable and the remainders of the classes are unified for each unifier of the two elements (step 6b).

3.8. Examples of semi-unification

The following are actual traces of the semi-unify, semi-unifyvector, semi-unifybag, and semi-unifyclass procedures as applied to the problem of unifying two expressions whose elements are respectively (1) A, ←X, ←←Y, and ←←Z and (2) A, B, B, and C. The semi-unification of vectors, bags, and classes, each consisting of this same collection of elements, illustrates the differences in the semi-unification algorithms employed for the different data types and the differing complexity of the problems as evidenced by the number of returned unifiers.

These examples do not illustrate all the features or the full complexity of the semi-unification algorithms since the second expression is variable free. Thus, the results are an instance of pattern matching as well as unification. The lack of variables (particularly the lack of fragment variables) in the second expression guarantees that the set of unifiers returned is a complete set, i.e., no other unifiers are possible.

The traced algorithms differ slightly from the preceding definitions affecting primarily the order in which operations are done.

Example 3.2.

```
ENTER UNIFYX:
!      X = (VECTOR A ←X ←←Y ←←Z)
!      Y = (VECTOR A B B C)
!   ENTER SEMIUNIFYVECTOR:
!   !      X = (A ←X ←←Y ←←Z)
!   !      Y = (A B B C)
!   !   ENTER SEMIUNIFYVECTOR:
!   !   !      X = (←X ←←Y ←←Z)
!   !   !      Y = (B B C)
!   !   !   ENTER UNIFYX:
!   !   !   !      X = ←X
!   !   !   !      Y = B
!   !   !   UNIFYX = (((X . B)))
!   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !      X = (←←Y ←←Z)
!   !   !   !      Y = (B C)
!   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !      X = (←←Z)
!   !   !   !   !      Y = (B C)
!   !   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !   !      X = NIL
!   !   !   !   !   !      Y = (B C)
!   !   !   !   !   SEMIUNIFYVECTOR = NIL
!   !   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !   !      X = NIL
!   !   !   !   !   !      Y = (C)
!   !   !   !   !   SEMIUNIFYVECTOR = NIL
!   !   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !   !      X = NIL
!   !   !   !   !   !      Y = NIL
!   !   !   !   !   SEMIUNIFYVECTOR = (((Z VECTOR B C) (Y VECTOR) (X . B)))
!   !   !   !   SEMIUNIFYVECTOR = (((Z VECTOR B C) (Y VECTOR) (X . B)))
!   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !      X = (←←Z)
!   !   !   !   !      Y = (C)
!   !   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !   !      X = NIL
!   !   !   !   !   !      Y = (C)
!   !   !   !   !   SEMIUNIFYVECTOR = NIL
!   !   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !   !      X = NIL
!   !   !   !   !   !      Y = NIL
!   !   !   !   !   SEMIUNIFYVECTOR = (((Z VECTOR C) (Y VECTOR B) (X . B)))
!   !   !   !   SEMIUNIFYVECTOR = (((Z VECTOR C) (Y VECTOR B) (X . B)))
!   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !      X = (←←Z)
!   !   !   !   !      Y = NIL
!   !   !   !   !   ENTER SEMIUNIFYVECTOR:
!   !   !   !   !   !      X = NIL
!   !   !   !   !   !      Y = NIL
!   !   !   !   !   SEMIUNIFYVECTOR = (((Z VECTOR) (Y VECTOR B C) (X . B)))
!   !   !   !   SEMIUNIFYVECTOR = (((Z VECTOR) (Y VECTOR B C) (X . B)))
!   !   !   SEMIUNIFYVECTOR = (((Z VECTOR B C) (Y VECTOR) (X . B))
                              ((Z VECTOR C) (Y VECTOR B) (X . B))
```

```
                                ((Z VECTOR) (Y VECTOR B C) (X . B)))
!  !  SEMIUNIFYVECTOR = (((Z VECTOR B C) (Y VECTOR) (X . B))
                         ((Z VECTOR C) (Y VECTOR B) (X . B))
                         ((Z VECTOR) (Y VECTOR B C) (X . B)))
!  SEMIUNIFYVECTOR = (((Z VECTOR B C) (Y VECTOR) (X . B))
                      ((Z VECTOR C) (Y VECTOR B) (X . B))
                      ((Z VECTOR) (Y VECTOR B C) (X . B)))
UNIFYX = (((Z VECTOR B C) (Y VECTOR) (X . B))
         ((Z VECTOR C) (Y VECTOR B) (X . B))
         ((Z VECTOR) (Y VECTOR B C) (X . B)))
```

Example 3.3.

```
ENTER UNIFYX:
!     X = (BAG A ←X ←←Y ←←Z)
!     Y = (BAG A B B C)
!  ENTER SEMIUNIFYBAG:
!  !     X = (A ←X ←←Y ←←Z)
!  !     Y = (A B B C)
!  !  ENTER UNIFYX:
!  !  !     X = ←X
!  !  !     Y = B
!  !  UNIFYX = (((X . B)))
!  !  ENTER SEMIUNIFYBAG:
!  !  !     X = (←←Y ←←Z)
!  !  !     Y = (B C)
!  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !     X = (←←Z)
!  !  !  !     Y = NIL
!  !  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !  !     X = NIL
!  !  !  !  !     Y = NIL
!  !  !  !  SEMIUNIFYBAG = (((Z BAG) (Y BAG B C) (X . B)))
!  !  !  SEMIUNIFYBAG = (((Z BAG) (Y BAG B C) (X . B)))
!  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !     X = (←←Z)
!  !  !  !     Y = (C)
!  !  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !  !     X = NIL
!  !  !  !  !     Y = NIL
!  !  !  !  SEMIUNIFYBAG = (((Z BAG C) (Y BAG B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !  !     X = NIL
!  !  !  !  !     Y = (C)
!  !  !  !  SEMIUNIFYBAG = NIL
!  !  !  SEMIUNIFYBAG = (((Z BAG C) (Y BAG B) (X . B)))
!  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !     X = (←←Z)
!  !  !  !     Y = (B)
!  !  !  !  ENTER SEMIUNIFYBAG:
!  !  !  !  !     X = NIL
!  !  !  !  !     Y = NIL
!  !  !  !  SEMIUNIFYBAG = (((Z BAG B) (Y BAG C) (X . B)))
!  !  !  !  ENTER SEMIUNIFYBAG:
```

```
! ! ! ! !     X = NIL
! ! ! ! !     Y = (B)
! ! ! !  SEMIUNIFYBAG = NIL
! ! ! SEMIUNIFYBAG = (((Z BAG B) (Y BAG C) (X . B)))
! ! ! ENTER SEMIUNIFYBAG:
! ! ! !     X = (↔Z)
! ! ! !     Y = (B C)
! ! ! !  ENTER SEMIUNIFYBAG:
! ! ! !  !     X = NIL
! ! ! !  !     Y = NIL
! ! ! !  SEMIUNIFYBAG = (((Z BAG B C) (Y BAG) (X . B)))
! ! ! !  ENTER SEMIUNIFYBAG:
! ! ! !  !     X = NIL
! ! ! !  !     Y = (C)
! ! ! !  SEMIUNIFYBAG = NIL
! ! ! !  ENTER SEMIUNIFYBAG:
! ! ! !  !     X = NIL
! ! ! !  !     Y = (B)
! ! ! !  SEMIUNIFYBAG = NIL
! ! ! !  ENTER SEMIUNIFYBAG:
! ! ! !  !     X = NIL
! ! ! !  !     Y = (B C)
! ! ! !  SEMIUNIFYBAG = NIL
! ! ! SEMIUNIFYBAG = (((Z BAG B C) (Y BAG) (X . B)))
! ! SEMIUNIFYBAG = (((Z BAG) (Y BAG B C) (X . B))
                  ((Z BAG C) (Y BAG B) (X . B))
                  ((Z BAG B) (Y BAG C) (X . B))
                  ((Z BAG B C) (Y BAG) (X . B)))
! ! ENTER UNIFYX:
! ! !     X = ↔X
! ! !     Y = C
! ! UNIFYX = (((X . C)))
! ! ENTER SEMIUNIFYBAG:
! ! !     X = (↔↔Y ↔↔Z)
! ! !     Y = (B B)
! ! !  ENTER SEMIUNIFYBAG:
! ! !  !     X = (↔↔Z)
! ! !  !     Y = NIL
! ! !  ENTER SEMIUNIFYBAG:
! ! !  !  !     X = NIL
! ! !  !  !     Y = NIL
! ! !  SEMIUNIFYBAG = (((Z BAG) (Y BAG B B) (X . C)))
! ! !  SEMIUNIFYBAG = (((Z BAG) (Y BAG B B) (X . C)))
! ! !  ENTER SEMIUNIFYBAG:
! ! !  !     X = (↔↔Z)
! ! !  !     Y = (B)
! ! !  ENTER SEMIUNIFYBAG:
! ! !  !  !     X = NIL
! ! !  !  !     Y = NIL
! ! !  SEMIUNIFYBAG = (((Z BAG B) (Y BAG B) (X . C)))
! ! !  ENTER SEMIUNIFYBAG:
! ! !  !  !     X = NIL
! ! !  !  !     Y = (B)
! ! !  SEMIUNIFYBAG = NIL
```

```
!   !   !  SEMIUNIFYBAG = (((Z BAG B) (Y BAG B) (X . C)))
!   !   !  ENTER SEMIUNIFYBAG:
!   !   !  !     X = (↔Z)
!   !   !  !     Y = (B B)
!   !   !  !  ENTER SEMIUNIFYBAG:
!   !   !  !  !     X = NIL
!   !   !  !  !     Y = NIL
!   !   !  !  SEMIUNIFYBAG = (((Z BAG B B) (Y BAG) (X . C)))
!   !   !  !  ENTER SEMIUNIFYBAG:
!   !   !  !  !     X = NIL
!   !   !  !  !     Y = (B)
!   !   !  !  SEMIUNIFYBAG = NIL
!   !   !  !  ENTER SEMIUNIFYBAG:
!   !   !  !  !     X = NIL
!   !   !  !  !     Y = (B B)
!   !   !  !  SEMIUNIFYBAG = NIL
!   !   !  SEMIUNIFYBAG = (((Z BAG B B) (Y BAG) (X . C)))
!   !   SEMIUNIFYBAG = (((Z BAG) (Y BAG B B) (X . C))
                        ((Z BAG B) (Y BAG B) (X . C))
                        ((Z BAG B B) (Y BAG) (X . C)))
!   SEMIUNIFYBAG = (((Z BAG) (Y BAG B C) (X . B))
                   ((Z BAG C) (Y BAG B) (X . B))
                   ((Z BAG B) (Y BAG C) (X . B))
                   ((Z BAG B C) (Y BAG) (X . B))
                   ((Z BAG) (Y BAG B B) (X . C))
                   ((Z BAG B) (Y BAG B) (X . C))
                   ((Z BAG B B) (Y BAG) (X . C)))
UNIFYX = (((Z BAG) (Y BAG B C) (X . B))
         ((Z BAG C) (Y BAG B) (X . B))
         ((Z BAG B) (Y BAG C) (X . B))
         ((Z BAG B C) (Y BAG) (X . B))
         ((Z BAG) (Y BAG B B) (X . C))
         ((Z BAG B) (Y BAG B) (X . C))
         ((Z BAG B B) (Y BAG) (X . C)))
```

Example 3.4.

```
ENTER UNIFYX:
!     X = (CLASS A ↔X ↔Y ↔Z)
!     Y = (CLASS A B C)
!  ENTER SEMIUNIFYCLASS:
!  !     X = (A ↔X ↔Y ↔Z)
!  !     Y = (A B C)
!  !  ENTER UNIFYX:
!  !  !     X = ↔X
!  !  !     Y = B
!  !  UNIFYX = (((X . B)))
!  !  ENTER SEMIUNIFYCLASS:
!  !  !     X = (↔Y ↔Z A B)
!  !  !     Y = (C A B)
!  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !     X = (↔Z B A C B A)
!  !  !  !     Y = (C B A)
!  !  !  !  ENTER SEMIUNIFYCLASS:
```

```
!  !  !  !  !     X = (A B C A B C)
!  !  !  !  !     Y = (A B C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS A B C) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C A B)
!  !  !  !  !     Y = (C A B)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS A B) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C A C)
!  !  !  !  !     Y = (B A C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS A C) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C A)
!  !  !  !  !     Y = (B C A)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS A) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C B C)
!  !  !  !  !     Y = (A B C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS B C) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C B)
!  !  !  !  !     Y = (A C B)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS B) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C C)
!  !  !  !  !     Y = (A B C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS C) (Y CLASS C B A) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (A B C)
!  !  !  !  !     Y = (A B C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS)) (Y CLASS C B A) (X . B)))
!  !  !  SEMIUNIFYCLASS = (((Z CLASS A B C) (Y CLASS C B A) (X . B))
                           ((Z CLASS A B) (Y CLASS C B A) (X . B))
                           ((Z CLASS A C) (Y CLASS C B A) (X . B))
                           ((Z CLASS A) (Y CLASS C B A) (X . B))
                           ((Z CLASS B C) (Y CLASS C B A) (X . B))
                           ((Z CLASS B) (Y CLASS C B A) (X . B))
                           ((Z CLASS C) (Y CLASS C B A) (X . B))
                           ((Z CLASS) (Y CLASS C B A) (X . B)))
!  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !     X = (↔Z B A C B)
!  !  !  !     Y = (A C B)
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A B C A)
!  !  !  !  !     Y = (B C A)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS B C A) (Y CLASS C B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A B C)
!  !  !  !  !     Y = (A B C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS B C) (Y CLASS C B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A B A)
!  !  !  !  !     Y = (C B A)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS B A) (Y CLASS C B) (X . B)))
```

```
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A B)
!  !  !  !  !     Y = (C A B)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS B) (Y CLASS C B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A C A)
!  !  !  !  !     Y = (B C A)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS C A) (Y CLASS C B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A C)
!  !  !  !  !     Y = (B A C)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS C) (Y CLASS C B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A A)
!  !  !  !  !     Y = (B C A)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS A) (Y CLASS C B) (X . B)))
!  !  !  !  ENTER SEMIUNIFYCLASS:
!  !  !  !  !     X = (B C A)
!  !  !  !  !     Y = (B C A)
!  !  !  !  SEMIUNIFYCLASS = (((Z CLASS) (Y CLASS C B) (X . B)))
!  !  !  SEMIUNIFYCLASS = (((Z CLASS B C A) (Y CLASS C B) (X . B))
                           ((Z CLASS B C) (Y CLASS C B) (X . B))
                           ((Z CLASS B A) (Y CLASS C B) (X . B))
                           ((Z CLASS B) (Y CLASS C B) (X . B))
                           ((Z CLASS C A) (Y CLASS C B) (X . B))
                           ((Z CLASS C) (Y CLASS C B) (X . B))
                           ((Z CLASS A) (Y CLASS C B) (X . B))
                           ((Z CLASS) (Y CLASS C B) (X . B)))
```

truncated here for brevity; total of 132 unifiers returned

3.9. Termination and soundness of semi-unification

Three properties are required of the unification procedures: termination, soundness, and completeness. The termination property assures us that the procedure terminates after a finite number of steps and incidentally also assures us that the number of unifiers returned is finite. The soundness property assures us that all the substitutions returned really do unify the two input expressions. The completeness property assures us a complete set of unifiers is returned, i.e., no unifier of the two

expressions is unrepresented by some member of the returned unifiers. The semi-unification procedures are incomplete by design; we will later prove that semi-unification in combination with variable splitting is complete.

Theorem 3.1. The procedure semi-unify is sound and always terminates.

Proof. Our proof relies on the fact that each principal step of the semi-unification procedures either reduces the number of variables occurring in the input expressions (while possibly increasing the total number of elements) or reduces the total number of elements. In the case of class semi-unification, the procedures reduce the number of previously unmatched variables or total number of previously unmatched elements. Thus, the number of variables in the two expressions is constantly declining, and the total number of (unmatched) elements in the expressions is reduced between removals of variables.

Induction will be performed on the tuple (n_1,n_2) where n_1 is the number of distinct variables occurring in the expressions and n_2 is the total number of elements occurring in the expressions excluding those already matched as elements of classes.

(m_1,m_2) is defined to be less than (n_1,n_2) (written $(m_1,m_2)<(n_1,n_2)$) if and only if m_1 is less than n_1, or m_1 is equal to n_1 and m_2 is less than n_2. Thus, the measure for one pair of expressions is less than that for another if the first has fewer variables, or the same number of variables but fewer elements.

$(n_1,n_2)=(0,0)$. If $n_2=0$, then neither expression has any (unmatched) elements. Thus, each must be an empty vector, an empty bag, or a class with all elements matched. If they are not both empty vectors, not both empty bags, or not both classes with all elements matched, then the empty set of substitutions is returned (semi-unify steps 5b, 6b, 7b). Thus, for this case semi-unify immediately terminates and is sound since no proposed unifiers are returned. If they are both empty vectors, both empty bags, or both classes with all elements matched (i.e., identical classes), the null substitution is returned as the only unifier (semi-unify step 1). Thus in this case also, semi-unify immediately terminates. It is also sound since the null substitution is being returned only if the two expressions are equal.

Assume semi-unify always terminates and is sound for all pairs of expressions with measure $(n_1,n_2)<(m_1,m_2)$. Prove it terminates and is sound for $(n_1,n_2)=(m_1,m_2)$. Assume $(n_1,n_2)=(m_1,m_2)$ for two expressions. Each of the input expressions may be a constant, simple variable, vector, bag, or class.

1. If both expressions are constants, either the empty set of substitutions is returned (semi-unify step 4) or the null substitution is returned as the only unifier (semi-unify step 1). In either case, semi-unify immediately terminates. A substitution, the null substitution, is returned only if the constants are equal. Therefore, semi-unify is sound for this case.

2. If one expression is a constant and the other is a simple variable, the substitution of the constant for the variable is returned as the only unifier (semi-unify steps 2a, 3a). Thus, semi-unify immediately terminates for this case and is sound since the instantiation of the variable by the returned substitution results in the constant of the other expression.

3. If both expressions are simple variables, either the null substitution is returned as the only unifier (semi-unify step 1) or the substitution of the second variable for the first is returned as the only unifier (semi-unify step 2a). In either case, semi-unify immediately terminates. Semi-unify is sound for this case since it only returns the null substitution if the two variables are identical, and if they are not identical the substitution of one for the other results in the two instantiated expressions being identical.

4. If one expression is a constant and the other is a vector, bag, or class, the empty set of substitutions is returned (semi-unify step 4). In this case, semi-unify immediately terminates and is sound since no unifiers are returned.

5. If one expression is a simple variable and the other is a vector, bag, or class, either the substitution of the vector, bag, or class for the variable is returned as the only unifier (semi-unify steps 2b1, 2c1, 2d1, 3b1, 3c1, 3d1) or the vector, bag, or class is unified with a vector, bag, or class with new fragment variable elements substituted for the simple variable argument (semi-unify steps 2b2, 2c2, 2d2, 3b2, 3c2, 3d2). In the

first case, semi-unify immediately terminates and is sound since the substitution of the vector, bag, or class for the variable results in the instantiated expressions being equal. The second case will be treated further in cases 7, 8, and 9. No extra difficulty with termination occurs here since the replacement of the simple variable by a vector, bag, or class can only occur once. Any unifier of the expression derived from the simple variable and the other expression is also a unifier of the simple variable and the other expression. Therefore the recursive call is sound.

6. If both expressions are vectors, bags, or classes, but not both of the same type, the empty set of substitutions is returned (semi-unify steps 5b, 6b, 7b). In this case, semi-unify immediately terminates and is sound since no substitutions are returned.

7. If both expressions are vectors, either (a) the null substitution is returned as the only unifier (semi-unify step 1), (b) the empty set of substitutions is returned (semi-unifyvector steps 1b, 2b), (c) two identical elements are removed and the remaining subvectors are semi-unified (semi-unifyvector step 3), (d) substitutions are made for a fragment variable and the instantiated remaining subvectors are semi-unified (semi-unifyvector steps 1a, 2a, 4a, 4b, 5a), or (e) two expressions are semi-unified and the instantiated remaining subvectors are semi-unified (semi-unifyvector step 5b).

7a. In subcase a, semi-unify immediately terminates and is sound since any substitution does unify identical expressions.

7b. In subcase b, semi-unifyvector immediately terminates and is sound since no substitutions are returned.

7c. In subcase c, semi-unifyvector is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 \leq n_2-2$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call terminates and is sound. Any unifier of two vectors is also a unifier of the same two vectors preceded by identical elements. Therefore, semi-unifyvector is sound for this subcase.

7d. In subcase d, semi-unifyvector is called recursively with expressions with measure (n_1-1,x) for some x. Thus, since $(n_1-1,x)<(n_1,n_2)$ for all x, by the inductive assumption this recursive call terminates and is sound and the returned substitutions unify the remainder vectors. The composition of the substitution of the initial subvector for the fragment variable and a returned unifier for the remaining vectors unifies the original vectors. Therefore, semi-unifyvector is sound for this subcase.

7e. In subcase e, semi-unify is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2<n_2$. If called at all (if semi-unify[a;b] succeeds), semi-unifyvector is called recursively with expressions with measure (x_1,x_2) with $x_1<n_1$ for some x_2 (since if the first elements are unifiable but not identical at least one variable must have been substituted for). Thus, by the inductive assumption, each recursive call terminates and is sound. The composition of the returned substitutions unify the unified remainder vectors preceded by the first elements and semi-unifyvector is sound for this subcase.

8. If both expressions are bags, either (a) the null substitution is returned as the only unifier (semi-unify step 1), (b) the empty set of substitutions is returned (semi-unifybag steps 1b, 2b), (c) two identical elements are removed and the remaining subbags are semi-unified (semi-unifybag step 3), (d) substitutions are made for a fragment variable and the instantiated remaining subbags are semi-unified (semi-unifybag steps 1a, 2a, 4a, 4b, 5a), or (e) two expressions are semi-unified and the instantiated remaining subbags are semi-unified (semi-unifybag step 5b).

8a. In subcase a, semi-unify immediately terminates and is sound since any substitution does unify identical expressions.

8b. In subcase b, semi-unifybag immediately terminates and is sound since no substitutions are returned.

8c. In subcase c, semi-unifybag is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 \leq n_2-2$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call terminates and is sound. Any unifier of two bags is also a unifier of the same two bags with identical elements added. Therefore, semi-unifybag is sound for this subcase.

8d. In subcase d, semi-unifybag is called recursively with expressions with measure (n_1-1,x) for some x. Thus, since $(n_1-1,x)<(n_1,n_2)$ for all x, by the inductive assumption this recursive call terminates and is sound. The composition of the substitution of the subbag for the fragment variable and a returned unifier for the remaining bags unifies the original bags. Therefore, semi-unifybag is sound for this subcase.

8e. In subcase e, semi-unify is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 < n_2$. If called at all (if semi-unify[a;b] succeeds), semi-unifybag is called recursively with expressions with measure (x_1,x_2) with $x_1 < n_1$ for some x_2 (since if the matched elements are unifiable but not identical at least one variable must have been substituted for). Thus, by the inductive assumption, each recursive call terminates and is sound. The composition of the returned substitutions unify the unified remainder bags with matched elements added and semi-unifybag is sound for this subcase.

9. If both expressions are classes, either (a) the null substitution is returned as the only unifier (semi-unify step 1), (b) one of two identical elements in a single class is removed and the remaining subclasses semi-unified (semi-unifyclass steps 1, 2), (c) two identical elements are matched against each other and the remaining subclasses are semi-unified (semi-unifyclass step 3), (d) substitutions are made for a fragment variable and the instantiated remaining subclasses are semi-unified (semi-unifyclass steps 5a, 5b, 6a), or (e) two expressions are semi-unified and the instantiated remaining subclasses are semi-unified (semi-unifyclass step 6b). The last two possibilities may occur after a one time interchange of the two classes (semi-unifyclass step 4).

9a. In subcase a, semi-unify immediately terminates and is sound since any substitution does unify identical expressions.

9b. In subcase b, semi-unifyclass is called recursively with

expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 \leq n_2-1$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call terminates and is sound. Any unifier of two classes is also a unifier of the same two classes with a duplicate element added to one of the classes. Therefore, semi-unifyclass is sound for this subcase.

9c. In subcase c, semi-unifyclass is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 \leq n_2-2$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call terminates and is sound. (Note. In this subcase, and in subcases d and e, semi-unifyclass is called recursively with the same expressions as its input except for being reordered and instantiated to reflect the matching performed. The reordering plus the restriction by step 4 of the procedure that pairs of elements in the identical (previously matched) tails of the two classes not be matched against each other assures termination. In computing (n_1,n_2) for these expressions, only variables and elements not contained in the identical tails are counted.) Since semi-unifyclass is called recursively with the identical expressions except for reordering to reflect restrictions on future matching, the unifiers returned by the recursive call unify the input expressions and semi-unifyclass is sound for this subcase.

9d. In subcase d, semi-unifyclass is called recursively with expressions with measure (n_1-1,x) for some x. Thus, since $(n_1-1,x)<(n_1,n_2)$ for all x, by the inductive assumption this

recursive call terminates and is sound. Since semi-unifyclass is called recursively with the identical expressions to its input except for instantiation by the substitution of the subclass for the fragment variable and reordering to reflect restrictions on future matching, the composition of the substitution of the subclass for the fragment variable and the unifiers returned by the recursive call unify the input expressions and semi-unifyclass is sound for this subcase.

9e. In subcase e, semi-unify is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 \leq n_2$. If called at all (if semi-unify[a;b] succeeds), semi-unifyclass is called recursively with expressions with measure (x_1,x_2) with $x_1 < n_1$ for some x_2 (since if the matched elements are unifiable but not identical at least one variable must have been substituted for). Thus, by the inductive assumption, each recursive call terminates and is sound. Since semi-unifyclass is called recursively with the identical expressions to its input except for instantiation by the unifier of the two matched elements and reordering to reflect restrictions on future matching, the composition of the unifiers returned by the recursive calls unify the input expressions and semi-unifyclass is sound for this subcase.

Thus, subject to the inductive assumption that semi-unify terminates and is sound for all pairs of expressions with measure $(n_1,n_2)<(m_1,m_2)$, semi-unify terminates and is sound for pairs of expressions with measure $(n_1,n_2)=(m_1,m_2)$.

Therefore, since semi-unify terminates and is sound for

$(n_1,n_2)=(0,0)$, semi-unify terminates and is sound for every pair of expressions. □

3.10. Variable splitting

Variable splitting is the process by which semi-unification can be extended to a complete unification algorithm.

Variable splitting consists of replacing single fragment variables by two newly introduced fragment variables. This introduction of additional fragment variables will, in general, result in the generation of additional unifiers by the semi-unification algorithm.

For example, consider the problem of unifying (VECTOR ↔X A) and (VECTOR A ↔X). These expressions have an infinite number of unifiers with X assigned vectors consisting of an arbitrary number of A's. However, the semi-unification algorithm returns only the unifiers with X assigned the empty vector and the vector consisting of a single A since fragment variables are constrained by the algorithm to match an integral number of elements in the opposing expression.

After substitution of (VECTOR ↔X1 ↔X2) for fragment variable X in the two expressions, the expressions are (VECTOR ↔X1 ↔X2 A) and (VECTOR A ↔X1 ↔X2). The semi-unification algorithm returns the unifiers {X1←(VECTOR),X2←(VECTOR)}, {X1←(VECTOR),X2←(VECTOR A)}, {X1←(VECTOR A),X2←(VECTOR)}, and {X1←(VECTOR A),X2←(VECTOR A)} for these expressions. The first

three of these unifiers duplicate the previous result of unifying (VECTOR ←←X A) and (VECTOR A ←←X), but the fourth is new representing the assignment of (VECTOR A A) to X.

It is clear that additional applications of variable splitting would successively result in the generation of the infinite sequence of unifiers of (VECTOR ←←X A) and (VECTOR A ←←X). For example, in the above expressions after variable splitting, if (VECTOR ←←X3 ←←X4) were substituted for variable X1 or X2 the additional unifier {X←(VECTOR A A A)} would be returned by the semi-unification algorithm.

Some constraints can be added to the semi-unification and variable splitting process to prevent the generation of or to detect and eliminate unifiers duplicating those generated prior to a variable splitting operation.

First, no variable introduced by variable splitting should be bound to the empty vector, bag, or class. If this occurred, the other variable introduced by variable splitting would be treated identically to the variable replaced by variable splitting in the previous semi-unification and the unifiers produced would duplicate unifiers generated before variable splitting.

This constraint is readily enforced by use of a simple syntactic device: using ←←← prefix variables rather than ←← prefix variables in variable splitting. In the example above, variable splitting would result in the formation of (VECTOR ←←←X1 ←←←X2 A) and (VECTOR A ←←←X1 ←←←X2) with only the unifier {X1←(VECTOR A),X2←(VECTOR A)} returned by the semi-unification

algorithm eliminating the generation of the three duplicate unifiers.

Second, pairs of variables introduced by a variable splitting operation should not collectively be assigned a fewer or equal number of elements to the maximum number of elements assigned by the semi-unification algorithm to the split variable before variable splitting. Restated, the split variable should be assigned a greater number of elements in the split case than in the unsplit case.

For example, consider the (degenerate) case of unifying (VECTOR ←X ←Y) and (VECTOR A B) resulting in the unifiers {X←(VECTOR),Y←(VECTOR A B)}, {X←(VECTOR A),Y←(VECTOR B)}, and {X←(VECTOR A B),Y←(VECTOR)} being returned by the semi-unification algorithm. Applying variable splitting to variable X of these two expressions, the expressions (using the ←←← prefix variable device suggested above) (VECTOR ←←←X1 ←←←X2 ←Y) and (VECTOR A B) are formed with unifier {X1←(VECTOR A),X2←(VECTOR B),Y←(VECTOR)} returned by the semi-unification algorithm. But this unifier duplicates the third unifier of the expressions before variable splitting, violating the condition that X1 and X2 be collectively assigned more than two elements.

The use of ←←← prefix variables in variable splitting is a special case of this constraint, requiring that the introduced variables be matched (if at all) to at least two elements.

Third, pairs of variables introduced by variable splitting must not be universally matched together against variables in the

opposing expression. Otherwise, they are being treated as a unit and their being derived from splitting a variable results in the generation of no additional unifiers.

For example, consider the case of (VECTOR ↔X ↔Y) and (VECTOR ↔U ↔V) resulting in the unifiers {X←(VECTOR),Y←(VECTOR ↔U ↔V)}, {X←(VECTOR ↔U),Y←(VECTOR ↔V)}, {X←(VECTOR ↔U ↔V),Y←(VECTOR)}, {U←(VECTOR),V←(VECTOR ↔X ↔Y)}, and {U←(VECTOR ↔X ↔Y),V←(VECTOR)} being returned by the semi-unification algorithm. Applying variable splitting to variable X of these two expressions, the expressions (VECTOR ↔↔X1 ↔↔X2 ↔Y) and (VECTOR ↔U ↔V) are formed with unifiers {X1←(VECTOR ↔U),X2←(VECTOR ↔V),Y←(VECTOR)}, {U←(VECTOR),V←(VECTOR ↔↔X1 ↔↔X2 ↔Y)}, {U←(VECTOR ↔↔X1),V←(VECTOR ↔↔X2 ↔Y)}, {U←(VECTOR ↔↔X1 ↔↔X2),V←(VECTOR ↔Y)}, and {U←(VECTOR ↔↔X1 ↔↔X2 ↔Y),V←(VECTOR)} returned by the semi-unification algorithm. The first of these unifiers is eliminable by the second constraint discussed above; the second, fourth, and fifth are eliminable by the current third constraint since ↔↔X1 and ↔↔X2 were always matched together; only the third is a new unifier.

3.11. Semi-unification and elaboration

It is useful to generalize the notions of semi-unification and variable splitting to create a theory of complete unification for complex data types. Such a theory would provide an approach to unification in more complex domains such as those employing

general pattern matching actors as in PLANNER [21]. In addition, it will aid in the proof of completeness for semi-unification and variable splitting.

Consider the case of pattern matching actors in PLANNER. These are to be distinguished from the more general actors consider in later work by Hewitt [23]. A (pattern matching) actor is a completely general function which matches expressions. Constants (which match themselves) and variables (which match their value or are assigned a value) are examples of actors. Vectors, bags, and classes are also actors which are restricted to matching vectors, bags, and classes respectively and have differing conventions on treatment of expressions in the matched expression, i.e., whether they are order independent and whether they may be matched to more than expression.

An example of a more complex actor is the following from [21]:

```
<define palindrome
        <actor []
                ;"palindrome is a actor of no arguments"
                <either
                        <empty>
                        ;"a palindrome is either empty or"
                        <declaration [x]
                                ;"declare a new local x"
                                <list  _x {palindrome} .x>
                                ;"let x be the first element of the
                                        linear structure.
                                        Also x must be the last element
```

with a palindrome
in between">>>>

The palindrome actor matches palindromes, sequences of expressions which read the same way backwards as forwards. It is composed of three additional actors: "either" which matches any expression which matches one of its arguments, "empty" which matches the empty expression, and "list" which matches any expression composed of its arguments in sequence. It also recursively calls itself in seeking to match an expression composed of a subexpression followed by a palindrome followed by the same subexpression.

Pattern matching is concerned only with the syntax of the matched expression. Pattern matching matches expressions without regard to the semantics of the matched expression. For example, the palindrome actor will examine expressions to determine if they read the same backwards as forwards, but will not attempt to find instances of the matched expression if it includes actors such as variables which are palindromes. The consideration of the semantics of the matched expression, and the attempt to find instances of the matched expression for which the pattern match would be successful, is the purpose of unification as opposed to pattern matching.

As a practical matter, it is not reasonable to define unification algorithms for large classes of actors. The definition of such procedures for constants, simple variables, fragment variables, vectors, bags, and classes presented here is

quite complex enough without the addition of more actors. Addition of another actor to the set of actors used to construct expressions for which unification algorithms have been devised requires explicit consideration of how the new actor is to be unified against each of the previously defined actors and against itself.

In addition to the practical impossibility of defining unification algorithms for large classes of actors, there are theoretical difficulties. If the actors are be sufficiently general, it is undecidable whether two expressions composed from the set of actors have any unifier at all. This can be seen as a consequence of the undecidability of the emptiness of the intersection of two context free languages. If the set of actors from which expressions can be constructed is sufficiently powerful to generate the set of context free languages, it is not possible to determine in general whether any pair of such expressions has a unifier.

The solution to the problem of unification with general actors is, rather than defining special unification algorithms for handling each of the actors (which in general will require consideration of each pair of actors, one appearing in each of the expressions being matched), to establish a convention concerning the form of expressions unifiable by a semi-unification algorithm with a completely general expression and generable by an elaboration algorithm from a completely general expression.

For example, the palindrome actor as written above is complete when matched against expressions composed entirely of constants. Thus, this actor could serve as a semi-unification procedure for palindromes, the matched expression being restricted to being composed of constants. It is also clear that a procedure could be written that successively produces all expressions composed of constants which are palindromes. Such a procedure would be called an elaboration procedure.

In generating a complete unification procedure for a set of actors, a semi-unification procedure and elaboration procedure would be defined for each actor, with the form of the matched expression in semi-unification and the form of the expression generated by elaboration both dictated by the commonly understood convention. In the extreme case, as suggested by the palindrome example above, the convention could be that the expression be composed solely of constants.

The procedure for unifying two expressions then becomes: use the elaboration procedures on one of the expression to form an instance of the expression in the expression sublanguage dictated by the convention, and match this newly formed expression against the other expression using the semi-unification procedures. In the example where the convention dictates that expressions be composed entirely of constants, the procedure becomes: form a ground instance of one of the expressions and match it to the other expression.

This procedure for unifying general expressions is complete

provided: every ground unifier of a general expression and an expression formed according to the convention is an instance of a unifier returned by the semi-unification procedures, and every instance of a general expression is an instance of an expression formed according to the convention by the elaboration procedures. Subject to the above two conditions, the statement of completeness means every ground unifier of two general expressions is an instance of a unifier returned by the semi-unification procedures applied to one of the expressions and an elaboration of the other expression.

The use of the set of expressions formed entirely of constants as the expression sublanguage understood by both the semi-unification and elaboration procedures is often more restrictive than necessary. A larger sublanguage which permits a substantial reduction in the number of expressions produced by elaboration is the language composed of constants and simple variables.

For example, consider again the palindrome actor. Assume it is intended to match a vector of elements. Then elaborations of the palindrome actor would be (using our notation for simple variables, etc.) (VECTOR), (VECTOR .X1 .X1), (VECTOR .X1 .X2 .X2 .X1), ... The semi-unification procedure for the palindrome actor would be required to match expressions consisting of constants and simple variables, assigning values to the simple variables as required to generate a unifier.

Constants and simple variables in this case are called

primitive actors, understood both by semi-unification and elaboration procedures. Fragment variables are non-primitive actors. Elaborations of a fragment variable would be the empty sequence, a single simple variable, two simple variables, etc. Elaborations of the expression (VECTOR ←←X ←←Y) would be (VECTOR), (VECTOR ←X1), (VECTOR ←Y1), (VECTOR ←X1 ←Y1), (VECTOR ←X1 ←X2), (VECTOR ←Y1 ←Y2), (VECTOR ←X1 ←X2 ←Y1), ...

In this case, unification of (VECTOR A ←←X) and (VECTOR ←←Y B) could result in (elaborating the second expression) unifying (VECTOR A ←←X) and (VECTOR B) with no unifiers, (VECTOR A ←←X) and (VECTOR ←Y1 B) with unifier (instantiating the parent fragment variable Y) {X←(VECTOR B),Y←(VECTOR A)}, (VECTOR A ←←X) and (VECTOR ←Y1 ←Y2 B) with unifier {X←(VECTOR ←Y2 B),Y←(VECTOR A ←Y2)}, (VECTOR A ←←X) and (VECTOR ←Y1 ←Y2 ←Y3 B) with unifier {X←(VECTOR ←Y2 ←Y3 B),Y←(VECTOR A ←Y2 ←Y3)}, etc.

3.12. Completeness of semi-unification and variable splitting

We will prove here that semi-unification is complete for two important cases: the case where one of the expressions is variable free and the case where both expressions are fragment variable free. The completeness of semi-unification for these two cases provides the basis for two uses of semi-unification and variable splitting which yield completeness in the general case.

The first use of semi-unification and variable splitting requires that variables in one of the expressions be instantiated

by expressions of the form (VECTOR ←←X), (BAG ←←X), and (CLASS ←←X) and variable splitting be applied to the result of such instantiations. The other expression is left unaltered.

The second use of semi-unification and variable splitting requires that variable splitting be applied to both expressions, but eliminates the need for instantiating either of them as in the first case.

For example, consider the unification of (CLASS ←X) and (CLASS (VECTOR A ←←Y) (VECTOR ←←Z B)). Applying the second method and performing a single variable splitting operation on one of the vectors in the second argument, we obtain (CLASS ←X) and (CLASS (VECTOR A ←←Y1 ←←Y2) (VECTOR ←←Z B)) with most general unifier {X←(VECTOR A ←←Y1 B), Y←(VECTOR ←←Y1 B), Z←(VECTOR A ←←Y1)}. In this example, we were successful in applying the method and using variable splitting on only one of the expressions; in general, this will not work and variable splitting must be applied to both expressions. An example for which variable splitting must be applied to both methods is (VECTOR (CLASS ←X) (CLASS (VECTOR A ←←V) (VECTOR ←←W B))) and (VECTOR (CLASS (VECTOR A ←←Y) (VECTOR ←←Z B)) (CLASS ←U)), i.e., an extension of the earlier problem where now fragment variables which must be split appear in both expressions.

Application of the first method and applying variable splitting to the second expression works just as for the second method. However, if in using the first method, it is desired that variable splitting be performed on the first expression, it

must first be instantiated. The ←X must be replaced by (VECTOR ←←X1) and X1 must be split twice yielding (CLASS (VECTOR ←←X2 ←←X3 ←←X4)). Semi-unification will now find the most general unifier of the two expressions. In general, in applying the first method, variables must be instantiated by expressions with new fragment variable arguments with the same type as the type of expressions to which they may be matched. In the example, ←X had to be instantiated by a vector since it was going to be matched to a vector.

The choice of method is dependent on the expected form of expressions to be unified as discussed below.

Theorem 3.2. The procedure semi-unify is complete if at least one of the input expressions contains no variables (the procedure semi-unify is complete for pattern matching).

Proof. Induction will again be performed on the tuple (n_1,n_2) where n_1 is the number of distinct variables occurring in the expressions and n_2 is the total number of elements occurring in the expressions excluding those already matched as elements of classes.

$(n_1,n_2)=(0,0)$. If $n_2=0$, then neither expression has any (unmatched) elements. Thus, each must be an empty vector, an empty bag, or a class with all elements matched. If they are not both empty vectors, not both empty bags, or not both classes with all elements matched, then the empty set of substitutions is returned (semi-unify steps 5b, 6b, 7b). Such pairs of expressions have no unifier and thus semi-unify is complete for

this case. If they are both empty vectors, both empty bags, or both classes with all elements matched (i.e., identical classes), the null substitution is returned as the only unifier (semi-unify step 1). Semi-unify is complete for this case since every unifier is an instance of the returned null substitution.

Assume semi-unify is complete for all $(n_1,n_2)<(m_1,m_2)$. Prove it is complete for $(n_1,n_2)=(m_1,m_2)$. Assume $(n_1,n_2)=(m_1,m_2)$ for two expressions. Each of the input expressions must be a constant, simple variable, vector, bag, or class.

1. If both expressions are constants, either the empty set of substitutions is returned (semi-unify step 4) or the null substitution is returned as the only unifier (semi-unify step 1). Semi-unify is complete in the first case since non-identical constants have no unifier. Semi-unify is complete in the second case since every unifier is an instance of the returned null substitution.

2. If one expression is a constant and the other is a simple variable, the substitution of the constant for the variable is returned as the only unifier (semi-unify steps 2a, 3a). The substitution of the constant for the variable is the only unifier of the two expressions, so semi-unify is complete for this case.

3. If both expressions are simple variables, either the null substitution is returned as the only unifier (semi-unify step 1) or the substitution of the second variable for the first is returned as the only unifier (semi-unify step 2a). In the first case, semi-unify is complete since every unifier is an instance

of the returned null substitution. In the second case, semi-unify is complete since the returned substitution of one variable for the other is maximally general with the two instantiated variables equal to one of the original variables.

4. If one expression is a constant and the other is a vector, bag, or class, the empty set of substitutions is returned (semi-unify step 4). Semi-unify is complete in this case since such pairs of expressions have no unifier.

5. If one expression is a simple variable and the other is a vector, bag, or class, either the substitution of the vector, bag, or class for the variable is returned as the only unifier (semi-unify steps 2b1, 2c1, 2d1, 3b1, 3c1, 3d1) or the vector, bag, or class is unified with a vector, bag, or class with new fragment variable elements substituted for the simple variable argument (semi-unify steps 2b2, 2c2, 2d2, 3b2, 3c2, 3d2). In the first case, semi-unify is complete since the returned substitution for the variable by the other expression is maximally general with the two instantiated expressions equal to one of the original expressions. The second case cannot occur by the assumption that only one of the expressions has any variables.

6. If both expressions are vectors, bags, or classes, but not both of the same type, the empty set of substitutions is returned (semi-unify steps 5b, 6b, 7b). In this case, semi-unify is complete since such pairs of expressions have no unifier.

7. If both expressions are vectors, either (a) the null

substitution is returned as the only unifier (semi-unify step 1), (b) the empty set of substitutions is returned (semi-unifyvector steps 1b, 2b), (c) two identical elements are removed and the remaining subvectors are semi-unified (semi-unifyvector step 3), (d) substitutions are made for a fragment variable and the instantiated remaining subvectors are semi-unified (semi-unifyvector steps 1a, 2a, 4a, 4b, 5a), or (e) two expressions are semi-unified and the instantiated remaining subvectors are semi-unified (semi-unifyvector step 5b).

7a. In subcase a, semi-unify is complete since every unifier is an instance of the returned null substitution.

7b. In subcase b, semi-unifyvector is complete since such pairs of expressions have no unifier.

7c. In subcase c, semi-unifyvector is called recursively with expressions with measure (x_1,x_2) with $x_1<n_1$ and $x_2<n_2-2$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call is complete. Every unifier of two vectors with identical first elements is a unifier of the two vectors with identical first elements removed. Therefore, semi-unifyvector is complete for this subcase.

7d. In subcase d, semi-unifyvector is recursively called with expressions with measure (n_1-1,x) for some x. Thus, since $(n_1-1,x)<(n_1,n_2)$ for all x, by the inductive assumption this recursive call is complete. The assignments to the fragment variable are complete since (1) every possible assignment of an initial subvector for the fragment variable is performed since

there are, by assumption, no (fragment) variables in the opposing vector and (2) the substitution of subvectors for fragment variables are maximally general. Therefore, the compositions of substitutions of initial subvectors for the fragment variable and returned unifiers for the remaining vectors are a complete set of unifiers for the vectors and semi-unifyvector is complete for this subcase.

7e. In subcase e, semi-unify is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 < n_2$. If called at all (if semi-unify[a;b] succeeds), semi-unifyvector is called recursively with expressions with measure (x_1,x_2) with $x_1 < n_1$ for some x_2 (since if the first elements are unifiable but not identical at least one variable must have been substituted for). Thus, by the inductive assumption, each recursive call is complete. Therefore, the compositions of unifiers of the first elements and unifiers of the remaining vectors are a complete set of unifiers for the vectors and semi-unifyvector is complete for this subcase.

8. If both expressions are bags, either (a) the null substitution is returned as the only unifier (semi-unify step 1), (b) the empty set of substitutions is returned (semi-unifybag steps 1b, 2b), (c) two identical elements are removed and the remaining subbags are semi-unified (semi-unifybag step 3), (d) substitutions are made for a fragment variable and the instantiated remaining subbags are semi-unified (semi-unifybag steps 1a, 2a, 4a, 4b, 5a), or (e) two expressions are

semi-unified and the instantiated remaining subbags are semi-unified (semi-unifybag step 5b).

8a. In subcase a, semi-unify is complete since every unifier is an instance of the returned null substitution.

8b. In subcase b, semi-unifybag is complete since such pairs of expressions have no unifier.

8c. In subcase c, semi-unifybag is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 \leq n_2-2$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call is complete. Every unifier of two bags with identical elements is a unifier of the two bags with identical elements removed. Therefore, semi-unifybag is complete for this subcase.

8d. In subcase d, semi-unifybag is recursively called with expressions with measure (n_1-1,x) for some x. Thus, since $(n_1-1,x)<(n_1,n_2)$ for all x, by the inductive assumption this recursive call is complete. The assignments to the fragment variable are complete since (1) every possible assignment of a subbag for the fragment variable is performed since there are, by assumption, no (fragment) variables in the opposing bag and (2) the substitution of subbags for fragment variables are maximally general. Therefore, the compositions of substitutions of subbags for the fragment variable and returned unifiers for the remaining bags are a complete set of unifiers for the bags and semi-unifybag is complete for this subcase.

8e. In subcase e, semi-unify is called recursively with expressions with measure (x_1,x_2) with $x_1 \leq n_1$ and $x_2 < n_2$. If called

at all (if semi-unify[a;b] succeeds), semi-unifybag is called recursively with expressions with measure (x_1,x_2) with $x_1<n_1$ for some x_2 (since if the elements are unifiable but not identical at least one variable must have been substituted for). Thus, by the inductive assumption, each recursive call is complete. In addition, the first element of the first bag is allowed to be matched against every element of the second bag. This is a complete set of matches for the first element of the first bag. Therefore, the compositions of unifiers of the matched elements and unifiers of the remaining bags are a complete set of unifiers for the bags and semi-unifybag is complete for this subcase.

9. If both expressions are classes, either (a) the null substitution is returned as the only unifier (semi-unify step 1), (b) one of two identical elements in a single class is removed and the remaining subclasses semi-unified (semi-unifyclass steps 1, 2), (c) two identical elements are matched against each other and the remaining subclasses are semi-unified (semi-unifyclass step 3), (d) substitutions are made for a fragment variable and the instantiated remaining subclasses are semi-unified (semi-unifyclass steps 5a, 5b, 6a), or (e) two expressions are semi-unified and the instantiated remaining subclasses are semi-unified (semi-unifyclass step 6b). The last two possibilities may occur after a one time interchange of the two classes (semi-unifyclass step 4).

9a. In subcase a, semi-unify is complete since every unifier is an instance of the returned null substitution.

9b. In subcase b, semi-unifyclass is called recursively with expressions with measure (x_1,x_2) with $x_1<n_1$ and $x_2<n_2-1$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call is complete. Every unifier of two classes with repeated elements in a class is a unifier of the two classes with all such elements but one in each class removed. Therefore, semi-unifyclass is complete for this subcase.

9c. In subcase c, semi-unifyclass is called recursively with expressions with measure (x_1,x_2) with $x_1\leq n_1$ and $x_2<n_2-1$. Thus, since $(x_1,x_2)<(n_1,n_2)$, by the inductive assumption this recursive call is complete. Every unifier of two classes with identical elements is a unifier of the two classes with identical elements removed. Therefore, semi-unifyclass is complete for this subcase.

9d. In subcase d, semi-unifyclass is recursively called with expressions with measure (n_1-1,x) for some x. Thus, since $(n_1-1,x)<(n_1,n_2)$ for all x, by the inductive assumption this recursive call is complete. The assignments to the fragment variable are complete since (1) every possible assignment of a subclass for the fragment variable is performed since there are, by assumption, no (fragment) variables in the opposing class and (2) the substitution of subclasses for fragment variables are maximally general. Therefore, the compositions of substitutions of subclasses for the fragment variable and returned unifiers for the remaining classes are a complete set of unifiers for the classes and semi-unifyclass is complete for this subcase.

9e. In subcase e, semi-unify is called recursively with expressions with measure (x_1,x_2) with $x_1<n_1$ and $x_2<n_2$. If called at all (if semi-unify[a;b] succeeds), semi-unifyclass is called recursively with expressions with measure (x_1,x_2) with $x_1<n_1$ for some x_2 (since if the first elements are unifiable but not identical at least one variable must have been substituted for). Thus, by the inductive assumption, each recursive call is complete. In addition, the first element of the first class is allowed to be matched against every element of the second class, i.e., every element of the original second class since even previously matched elements are retained. This is a complete set of matches for the first element of the first class. Therefore, the compositions of unifiers of the matched elements and unifiers of the remaining classes are a complete set of unifiers for the classes and semi-unifyclass is complete for this subcase.

Thus, subject to the inductive assumption that semi-unify is complete in the case where at least one of the input expressions contains no variables for all expressions with measure $(n_1,n_2)<(m_1,m_2)$, semi-unify is complete for all expressions at least one of which contains no variables with measure $(n_1,n_2)=(m_1,m_2)$. □

The proof presented here is actually a little stronger (in a difficult way to state or use) than that needed. The assumption that one of the expressions has no variables is only used in steps 5, 7d, 8d, and 9d of the proof; it is not used in some other steps in which variables could be in both expressions.

The assumption is required in steps 7d, 8d, and 9d because semi-unification does not permit an initial subvector, a subbag, or a subclass including only part of a fragment variable to be assigned to a fragment variable; only an integral number of elements can be assigned, i.e., a fragment variable in the expression opposing the fragment variable being substituted for must be either totally included in or excluded from the substitution.

This theorem is adequate for proving the completeness of a primitive form of semi-unification and elaboration. The semi-unification algorithm is complete if at least one expression has no variables; the elaboration process produces all ground instances of one of the expressions. By the completeness of semi-unification in the restricted case, and the exhaustiveness of elaboration, semi-unification plus elaboration is complete.

However, this form of elaboration is primitive and inefficient. We choose to use a more powerful elaboration process (variable splitting). The result of variable splitting is so general, retaining all the kinds of elements as the original expression, and retaining fragment variables in particular, that semi-unification applied to two expressions, one modified by variable splitting, is incomplete in general. Nevertheless, semi-unification plus variable splitting is complete since unifiers not returned by semi-unification after a sequence of variable splitting operations will be returned by semi-unification after a different sequence of variable splitting operations.

Theorem 3.3. The unifiers returned by semi-unification of two expressions are instances of unifiers returned by semi-unification of generalizations of the two expressions where individual constants or simple variables are replaced by fragment variables.

Proof. The point of this theorem is that anything you can do with constants or simple variables in terms of unification you can do with a fragment variable. Note that this theorem is not true where the generalization associates a fragment variable with a sequence of simple variables or constants of length greater than one; if this were so, semi-unification alone would be complete and variable splitting would be unnecessary.

A simple variable or constant can be matched in three ways: (1) it can be matched against another simple variable or constant, (2) it can be included in the value of a fragment variable, or (3) it can be an element of an expression matched to a simple variable or included in the value of a fragment variable.

In the first case, a generalization where the simple variable or constant is replaced by a fragment variable could be successfully matched by matching the fragment variable to the other simple variable or constant.

In the second and third cases, a generalization where the simple variable or constant is replaced by a fragment variable could be successfully matched in precisely the same way as before.

Therefore, no unifiers are lost by the use of fragment variables rather than simple variables or constants. □

Theorem 3.4. The use of the procedure semi-unify with variable splitting applied to instances of one of the expressions is complete where the instances are formed by replacing simple or fragment variables by vectors, bags, or classes with new fragment variable arguments.

Proof. True by the completeness of semi-unification for pattern matching, i.e., the completeness of semi-unification where at least one of the expressions has no variables, and the previous theorem that generalizing simple variables or constants to fragment variables results in the loss of no unifiers. The procedure is just doing elaboration on one of the expressions except instead of producing ground expressions, its output is expressions with fragment variables which could represent constants. □

Theorem 3.5. The procedure semi-unify is complete if neither of the input expressions has any fragment variables.

Proof. The proof is essentially the same as that for the earlier theorem stating the completeness of semi-unification for pattern matching, i.e., the completeness of semi-unification in the case where at least one expression has no variables.

The only differences are (1) all references to fragment variables (cases 7d, 8d, 9d) may be omitted and (2) the second part of case 5 (the expression being matched to a simple variable

containing the variable) may now occur. In case 5, since there are no fragment variables in the two expressions, the occurrence of a variable in the opposing expression in case 5 must be a simple variable occurrence. But such an expression cannot be unified with the simple variable, and thus semi-unify is complete for this case. The introduction of fragment variables in this case (by semi-unify steps 2b2, 2c2, 2d2, 3b2, 3c2, 3d2) presents no difficulty since (by soundness) the unification will fail, and the introduced fragment variables will not be available for future recursive calls to violate the assumption of the expressions not having any fragment variables. □

Theorem 3.6. The use of the procedure semi-unify with variable splitting applied to both expressions is complete.

Proof. By the previous theorem, semi-unification is complete for expressions having no fragment variables. Thus, replacing fragment variables in both input expressions by sequences of distinct simple variables plus semi-unification is complete due to the completeness of semi-unification for the altered expressions and the exhaustiveness of the replacement of fragment variables by sequences of simple variables (we are just elaborating both input expressions to expressions with no fragment variables by replacing them by sequences of simple variables).

By the theorem that the unifiers returned by semi-unification of two expressions are instances of unifiers returned by semi-unification of generalizations of the two expressions where

individual constants or simple variables are replaced by fragment variables, replacing fragment variables in both input expressions by sequences of distinct fragment (rather than simple) variables plus semi-unification is complete. But such sequences may be generated by applying variable splitting, so semi-unification plus the application of variable splitting to both expressions is complete. □

We now have two alternative methods of extending semi-unification to completeness by use of variable splitting. Some choice must be made of whether to implement the first method, the second method, or both methods.

If, for example, it is known that all the expressions that will be matched against each other are one level deep (contain as elements only fragment variables and atomic expressions), then it is sensible to implement only the first method since no instantiation would be required and the use of variable splitting on the second expression would be redundant.

On the other hand, if it is known that the expressions that will be matched against each other will usually be quite complex with many levels, the required instantiation of the first method may be too inconvenient and costly to perform. In this case, the second method would probably be preferred.

A third alternative is to implement both methods and dynamically choose which to employ when presented with two expressions to be matched. For the completely general case where there is no a priori reason to expect expressions of one kind as

opposed to that of another, this is probably the best alternative permitting the more efficient use of variable splitting in the separate cases.

3.13. Vector unification

It has already been stated several times that (unless some extended notation for unifiers is adopted) the number of unifiers in the general vector unification case may be infinite.

For expository purposes, we present here a vector unification algorithm which, although possibly non-terminating even in cases where there are no unifiers, is sound and complete.

This vector unification algorithm illuminates the role of variable splitting in extending semi-unification to completeness, and is a model for the class unification algorithm to be presented later.

The unifyvector procedure, when presented with two non-empty vectors, always removes the first element of the first vector at each step.

When presented with two vectors whose first elements are not fragment variables, it directly unifies the first elements and then unifies the instantiated remainders of the vectors. When one or both of the first elements of the vectors are fragment variables, the substitution of the empty vector for the fragment variable and a general substitution including the opposing first element for the fragment variable are both tried.

Steps are numbered in close correspondence to the numbering in the vector semi-unification algorithm for comparison purposes.

unifyvector[x; y]=

1. If x is of the form (VECTOR) and
   a. y is of the form (VECTOR ↔Y β) then let θ={Y←(VECTOR)} and return {θσ | σ ∈ semi-unify[x; (VECTOR β)θ]}.
   b. y is not of the form (VECTOR ↔Y β) then return {}.
2. If y is of the form (VECTOR) and
   a. x is of the form (VECTOR ↔X α) then let θ={X←(VECTOR)} and return {θσ | σ ∈ semi-unify[(VECTOR α)θ; y]}.
   b. x is not of the form (VECTOR ↔X α) then return {}.
3. If x is of the form (VECTOR a α) and y is of the form (VECTOR a β) for arbitrary element a then semi-unify[(VECTOR α); (VECTOR β)].
4. If x is of the form (VECTOR ↔X α) then
   a. if y is of the form (VECTOR ↔Y β) then let θ={Y←(VECTOR ↔X ↔Y')} and return {θσ | σ ∈ unify[(VECTOR α)θ; (VECTOR ↔Y' β)θ]} and
   b. if y is of the form (VECTOR b β) and X is not contained in b then let θ={X←(VECTOR b ↔X')} and return {θσ | σ ∈ unify[(VECTOR ↔X' α)θ; (VECTOR β)θ]} and
   c. if y is of the form (VECTOR b β) where element b is not a fragment variable let θ={X←(VECTOR)} and return {θσ | σ ∈ unify[(VECTOR α)θ; yθ]}.
5. If x is of the form (VECTOR a α) where element a is not a fragment variable then

a. y is of the form (VECTOR ↔Y β) and Y is not contained in a then let θ={Y←(VECTOR a ↔Y')} and return {θσ | σ ∈ unify[(VECTOR α)θ; (VECTOR ↔Y' β)θ]} and

b. if y is of the form (VECTOR b β) where element b is not a fragment variable then return {θσ | θ ∈ unify[a; b] ∧ σ ∈ unify[(VECTOR α)θ; (VECTOR β)θ]} and

c. if y is of the form (VECTOR ↔Y β) then let θ={Y←(VECTOR)} and return {θσ | σ ∈ unify[xθ; (VECTOR β)θ]}.

Notes:

1. If one of the vectors is the empty vector and the other is not, the latter must be composed solely of fragment variables which must be bound to the empty vector. Otherwise, there is no unifier (steps 1, 2).

2. If the first elements of two vectors are identical, the remainders of the vectors are unified (step 3).

3. If the first element of one or both of the vectors is a fragment variable, the substitution for the fragment variable of the vector consisting of the opposing first element and a new fragment variable is performed (meaning the fragment variable includes the opposing first element) and the remainders of the vectors are unified (steps 4a, 4b, 5a).

4. If the first element of one (but not both) of the vectors is a fragment variable, the substitution of the empty vector for the fragment variable is performed and the remainders of the vectors are unified (steps 4c, 5c).

5. If neither of the first elements of the vectors is a

fragment variable, the two first elements are unified against each other and the remainders of the vectors are unified for each unifier of the first elements (step 5b).

This algorithm can readily be made substantially more efficient. One major cause of inefficiency is that every time a fragment variable is matched against a non-fragment variable, a new fragment variable is formed (steps 4b, 5a). Actually, it would be sufficient to match the fragment variable against an entire sequence of non-fragment variables similarly to semi-unification. Thus, when matching (VECTOR ↔X α) and (VECTOR A B ↔Y), the substitutions {X←(VECTOR)}, {X←(VECTOR A)}, and {X←(VECTOR A B ↔X1)} would be tried in parallel rather than the successive {X←(VECTOR)}, {X←(VECTOR A ↔X1)}, {X←(VECTOR A), X1←(VECTOR)}, and {X←(VECTOR A B ↔X2), X1←(VECTOR B ↔X2)}.

The algorithm is expressed in this simpler, less efficient form so it can serve as a template for a later description of the similar class unification algorithm for which similar fixes are difficult to describe and the less efficient form is therefore presented.

The algorithm can readily be shown to be sound using a proof similar to the soundness proof for (vector) semi-unification.

Assuming the completeness of the unify[a; b] call in step 5b, the algorithm is clearly complete (capable of generating all unifiers). There is no difficulty in establishing completeness in an induction proof where neither first element is a fragment variable. In the critical cases where either first element is a

fragment variable, the algorithm is exhaustive, trying both cases: the opposing element is or is not included in the substitution for the fragment variable.

One little trick in the algorithm is the elimination of the substitution of the empty vector for a fragment variable first element if the opposing first element is also a fragment variable. This is justifiable (doesn't result in loss of completeness) since the possibility is implicitly included in the substitution for the opposing fragment variable.

For example, in unifying (VECTOR ↔X α) and (VECTOR ↔Y β), the untried substitution {X←(VECTOR)} (and unification of (VECTOR α) and (VECTOR ↔Y β)) is an instance (with variable renaming (Y' for Y)) of the substitution {Y←(VECTOR ↔X ↔Y')} (and unification of instances of (VECTOR α) and (VECTOR ↔Y' β)) which is used. The trick merely postpones the trial assignment of the empty vector to the fragment variable until the fragment variable again appears first in a vector, and the opposing vector is either empty or has a first element which is not a fragment variable.

The reason this algorithm does not always terminate is that a fragment variable may be matched against the first element of the opposing vector spawning a new fragment variable indefinitely. Technically, the semi-unification termination proof breaks down when applied to this algorithm because the measure (n_1,n_2) of variables and elements does not always decrease when substitutions are made for fragment variables.

An example of the application of this algorithm is the unification of (VECTOR ↔X A) and (VECTOR A ↔X). A truncated trace follows.

```
unifyvector[(VECTOR ↔X A); (VECTOR A ↔X)]
|  unifyvector[(VECTOR A); (VECTOR A)]
|  return {{X←(VECTOR)}}
|  unifyvector[(VECTOR ↔X1 A); (VECTOR A ↔X1)]
|  |  unifyvector[(VECTOR A); (VECTOR A)]
|  |  return {{X←(VECTOR A); X1←(VECTOR)}}
|  |  unifyvector[(VECTOR ↔X2 A); (VECTOR A ↔X2)]
|  |  |  unifyvector[(VECTOR A); (VECTOR A)]
|  |  |  return {{X←(VECTOR A A), X1←(VECTOR A), X2←(VECTOR)}}
|  |  |  unifyvector[(VECTOR ↔X3 A); (VECTOR A ↔X3)]

...
```

A pathological example of the application of this algorithm is the unification of (VECTOR ↔X B) and (VECTOR A ↔X) which has no unifier. The algorithm does exactly as before, substituting for X either the empty vector or the vector consisting of A and a new fragment variable at each step. However, in this case unification of (VECTOR B) and (VECTOR A) fails and the unification algorithm finds no unifier (it is, after all, sound) and never terminates.

The use of a finite number of variable splitting operations and the application of the semi-unification algorithm effectively

imposes a limit on the number of times fragment variables are permitted to be matched and form new fragment variables. Variable splitting externalizes and bounds the variable splitting which occurs inside the vector unification algorithm.

3.14. Bag unification

In the case of bags, there does exist a complete unification algorithm.

In discussing this algorithm, we will assume that the two bags have no common elements. This presents no difficulty since if two bags do have common elements, no unifiers are lost and efficiency is gained if common elements are eliminated from the bags in pairs until no common elements remain.

The bag unification algorithm will be expressed in terms of an algorithm for the complete unification of bags composed only of fragment variables. The result of unifying such bags is an assignment to each variable of the bags a bag composed of some sequence of elements. For such an assignment to be a unifier, the only requirement is that for each element used in any assignment there are the same number of occurrences of that element in each of the instantiated bags. For example, in unifying (BAG ↔X ↔X ↔X ↔Y ↔Y) and (BAG ↔U ↔U ↔U ↔U ↔U ↔U ↔U ↔V ↔V ↔V ↔V ↔V), if element a is part of some assignment to one of the variables X, Y, U, or V, then 3 times the number of occurrences of a in the assignment for X plus 2

times the number of occurrences of a in the assignment for Y must equal 7 times the number of occurrences of a in the assignment for U plus 5 times the number of occurrences of a in the assignment for V.

With each pair of bags composed only of fragment variables is associated a single equation representing the number and multiplicity of variables in each bag. For example, the equation $3x+2y = 7u+5v$ is associated with the pair of bags given above. This equation succintly represents the condition for a substitution to be a unifier: that the sum of the number of occurrences of any element a in each variable multiplied by the multiplicity of the variable in the bag must be equal for the two bags.

Non-negative integral solutions to such equations can be used to represent unifiers. The solutions must be non-negative since each variable in a bag must be assigned a non-negative number of occurrences of each element; it cannot, for example, have -1 occurrences of the constant A. Likewise, the solutions considered must be integral since fractions of elements are not permitted in variable assignments. We will now concentrate on finding an adequate set of non-negative integral solutions to such equations.

Let us consider the equation $3x+2y = 7u+5v$. Solutions will be expressed in terms of a 4-tuple: (a,b,c,d) where a, b, c, and d are all non-negative integers and $3a+2b=7c+5d$, i.e., a, b, c, and d are non-negative integers which, when assigned to variables x, y, u, and v respectively, result in the equation being satisfied.

Immediately discoverable solutions to the equation are (7,0,3,0), (5,0,0,3), (0,7,2,0), and (0,5,0,2) obtained by taking pairs of variables from the two sides of the equation, setting all the other variables to zero, and giving the least non-negative integral solution (excluding the trivial solution of all zeroes). Other solutions are possible such as (4,1,2,0) and any sum of known solutions. What is needed is a finite set of solutions which expresses the set of all solutions.

If we were concerned with all real solutions to the equation, these could be obtained by taking linear combinations of any m+n-1 (where m is the number of variables in the left side of the equation, n is the number of variables in the right side of the equation) linearly independent solutions of the equation. m+n-1 linearly independent solutions are included among the m*n solutions generated by taking pairs of variables from the two sides and solving the resulting equation after setting all the other variables to zero.

This is not adequate for generating all the non-negative integral solutions by addition alone however (we are concerned with finding solutions generated as sums to satisfy the non-negativity and integrality constraints). For example, the solution (4,1,2,0) is not the result of adding any number of the pairwise solutions together. Thus, any set of m+n-1 linearly independent solutions must be augmented by additional non-negative integral solutions so that all such solutions may be generated.

We hypothesize here that every non-negative integral solutions can be formed by addition alone (or equivalently, by multiplication by a positive integer and addition) from the set of all non-negative integral solutions for which the value of the left side of the equation (equals the right side of the equation, and is called the value of the equation) is less than or equal to the maximum of the least common multiples of the coefficients of a variable on the left side and a variable on the right side.

The value of the equation 3x+2y = 7u+5v is 21 (3*7). The set of all non-negative integral solutions to the equation with value less than or equal to 21, excluding (0,0,0,0) (which is generable as the empty sum of solutions) and solutions composable from other solutions with value less than or equal to 21 is (1,1,0,1), (1,2,1,0), (0,5,0,2), (4,0,1,1), (0,6,1,1), (4,1,2,0), (0,7,2,0), (5,0,0,3), and (7,0,3,0). According to the hypothesis, all non-negative integral solutions to the equation are composable by addition from this set of 9 solutions.

Now let us return to the original bag unification problem which spawned this particular equation. Each solution represents a solution to the original bag unification problem. For example, (4,1,2,0) represents the solution {X←(BAG ↔T ↔T ↔T ↔T), Y←(BAG ↔T), U←(BAG ↔T ↔T), V←(BAG)} for arbitrary new variable T and all its instances. Thus we see that each solution can represent a unifier of the bags using a particular newly formed fragment variable.

Since fragment variables can be empty, we can freely compose a

unifier incorporating all the solutions. Let us do so. We first associate a new fragment variable with each solution:

(X,Y,U,V)

(1,1,0,1) ←T1

(1,2,1,0) ←T2

(0,5,0,2) ←T3

(4,0,1,1) ←T4

(0,6,1,1) ←T5

(4,1,2,0) ←T6

(0,7,2,0) ←T7

(5,0,0,3) ←T8

(7,0,3,0) ←T9

Then the first variable is assigned a bag composed of fragment variables in the number specified by the first components of the tuple, the second variable is assigned a bag composed of fragment variables in the number specified by the second component, etc. Thus, X would be assigned a bag with 1 T1, 1 T2, 4 T4's, 4 T6's, 5 T8's, and 7 T9's. The final, complete unifier is {X←(BAG ←T1 ←T2 ←T4 ←T4 ←T4 ←T4 ←T6 ←T6 ←T6 ←T6 ←T8 ←T8 ←T8 ←T8 ←T8 ←T9 ←T9 ←T9 ←T9 ←T9 ←T9 ←T9), Y←(BAG ←T1 ←T2 ←T2 ←T3 ←T3 ←T3 ←T3 ←T3 ←T5 ←T5 ←T5 ←T5 ←T5 ←T5 ←T6 ←T7 ←T7 ←T7 ←T7 ←T7 ←T7 ←T7), U←(BAG ←T2 ←T4 ←T5 ←T6 ←T6 ←T7 ←T7 ←T9 ←T9 ←T9), V←(BAG ←T1 ←T3 ←T3 ←T4 ←T5 ←T8 ←T8 ←T8)}. Each instantiated bag has 5 T1's, 7 T2's, 10 T3's, 12 T4's, 12 T5's, 14 T6's, 14 T7's, 15 T8's, and 21 T9's, 110 fragment variable occurrences in all.

The important point is that the single unifier presented above is the most general unifier; every unifier of the two bags in an instance of this unifier.

To be more precise in the definition of the algorithm, the algorithm consists of the following steps:

1. Form an equation from the two bags wnere the coefficient of each variable in the equation is equal to the multiplicity of the corresponding fragment variable.
2. Generate all non-negative integral solutions of the equation with value less than or equal to the maximum least common multiple of a coefficient from the left side and a coefficient from the right side of the equation.
3. Eliminate all solutions composable by addition from other solutions. Such solutions can be found by subtracting from them each solution with lesser value; if any difference is non-negative integral, the solution is composable from other solutions and can thus be rejected.
4. Associate with each solution a single new fragment variable.
5. Assemble a unifier composed of assignments to the original fragment variables of bags with as many of each new fragment variable as specified by the solution element associated with the original fragment variable and the new fragment variable.

This solution must be generalized in some way. Although the case of unifying bags composed solely of fragment variables is important, it is not sufficient.

There does not appear to be any straightforward method of

performing operations on a pair of bags to eliminate all the elements which are not fragment variables (generating partial unifiers in the process) so that the above algorithm is applicable to the preprocessed input.

One possible method is to merely perform all the possible unifications involving elements which are not fragment variables first. For example, consider unifying (BAG ↔X ↔Y) and (BAG A ↔Z ↔W). The constant A must be an element of either X or Y. Thus, {X←(BAG A ↔T1)} and {Y←(BAG A ↔T2)} (for new fragment variables T1 and T2) are legitimate partial unifiers of the two bags representing respectively that X contains A with remainder T1 and Y contains A with remainder T2. The instantiated bag pairs with matched elements removed are then (BAG ↔T1 ↔Y) and (BAG ↔Z ↔W), and (BAG ↔X ↔T2) and (BAG ↔Z ↔W) to which the above algorithm could be applied.

However, this method quickly breaks down. Consider the bags (BAG A ↔X ↔X) and (BAG ↔Y ↔Y) which, incidentally, have no unifier since the first bag clearly must have an odd number of A's and the second bag must have an even number of A's. We make the substitution {Y←(BAG A ↔T1)}, the only possible substitution since, if the expressions are unifiable, A must be an element of Y. After instantiating the expressions and removing the matched elements, the bags (BAG ↔X ↔X) and (BAG ↔T1 A ↔T1) which, given the commutativity of the bag data type and the commutativity of the unification operation, poses a unification problem identical to the original one. Thus, some other approach is needed.

The most straightforward approach which works is the following: from an arbitrary pair of bags to be unified form a new pair of bags in which each element in the original pair of bags which is not a fragment variable is replaced by a new fragment variable. For example, the bags (BAG ←←X ←←Y) and (BAG ←←S ←←Z ←←W) would be formed from the bags (BAG ←←X ←←Y) and (BAG A ←←Z ←←W) by replacing each occurrences of A by new fragment variable S.

Now solve the equation associated with the new pair of bags. For this problem, the associated equation (x+y=s+z+w) has the following set of solutions and associated fragment variables:

(X,Y,S,Z,W)

(1,0,1,0,0) ←←T1

(1,0,0,1,0) ←←T2

(1,0,0,0,1) ←←T3

(0,1,1,0,0) ←←T4

(0,1,0,1,0) ←←T5

(0,1,0,0,1) ←←T6

The resulting unifier is thus

{X←(BAG ←←T1 ←←T2 ←←T3), Y←(BAG ←←T4 ←←T5 ←←T6),
S←(BAG ←←T1 ←←T4), Z←(BAG ←←T2 ←←T5), W←(BAG ←←T3 ←←T6)}.

The final step of the algorithm is to unify in every possible way the elements replaced by fragment variables (after instantiating them by the returned unifier of the derived bags in case they contain occurrences of variables the same as the

unified fragment variables) with the values of the fragment variables returned by the algorithm for bags composed entirely of fragment variables. This unification presents no difficulty since one of the expressions is guaranteed to have no fragment variables at the top level by construction; thus, semi-unification will suffice here. In the example, the value of S is unified with (BAG A). This results in unifiers {T1←(BAG A), T4←(BAG)} and {T1←(BAG), T4←(BAG A)}.

Thus the final set of unifiers for the original two expressions consists of {X←(BAG A ↔T2 ↔T3), Y←(BAG ↔T5 ↔T6), Z←(BAG ↔T2 ↔T5), W←(BAG ↔T3 ↔T6)} and {X←(BAG ↔T2 ↔T3), Y←(BAG A ↔T5 ↔T6), Z←(BAG ↔T2 ↔T5), W←(BAG ↔T3 ↔T6)}.

Another example of the use of this bag unification algorithm is the unification of (BAG ←X ←Y) and (BAG A B).

Replacing each non-fragment variable element of the bags by a new fragment variable, we obtain (BAG ↔S1 ↔S2) and (BAG ↔S3 ↔S4). The values of S1, S2, S3, and S4 will later be matched to (BAG ←X), (BAG ←Y), (BAG A), and (BAG B) respectively.

The corresponding equation is $s_1+s_2=s_3+s_4$ with solutions and corresponding newly created fragment variables

```
(S1,S2,S3,S4)
(1,0,1,0)    ↔T1
(1,0,0,1)    ↔T2
(0,1,1,0)    ↔T3
(0,1,0,1)    ↔T4.
```

Thus, the general unifier of (BAG ↔S1 ↔S2) and (BAG ↔S3 ↔S4) is

```
{S1←(BAG ↔T1 ↔T2),
 S2←(BAG ↔T3 ↔T4),
 S3←(BAG ↔T1 ↔T3),
 S4←(BAG ↔T2 ↔T4)}.
```

Unifying the value of S1 ((BAG ←←T1 ←←T2)) with (BAG ←X), we obtain

```
{{S1←(BAG ←X),
  S2←(BAG ←←T3 ←←T4),
  S3←(BAG ←X ←←T3),
  S4←(BAG ←←T4),
  T1←(BAG ←X),
  T2←(BAG)},
 {S1←(BAG ←X),
  S2←(BAG ←←T3 ←←T4),
  S3←(BAG ←←T3),
  S4←(BAG ←X ←←T4),
  T1←(BAG),
  T2←(BAG ←X)}}.
```

Unifying the value of S2 ((BAG ←←T3 ←←T4)) with (BAG ←Y), we obtain

```
{{S1←(BAG ←X),
  S2←(BAG ←Y),
  S3←(BAG ←X ←Y),
  S4←(BAG),
  T1←(BAG ←X),
  T2←(BAG),
  T3←(BAG ←Y),
  T4←(BAG)},
 {S1←(BAG ←X),
  S2←(BAG ←Y),
  S3←(BAG ←X),
  S4←(BAG ←Y),
  T1←(BAG ←X),
  T2←(BAG),
  T3←(BAG),
  T4←(BAG ←Y)},
 {S1←(BAG ←X),
  S2←(BAG ←Y),
  S3←(BAG ←Y),
  S4←(BAG ←X),
  T1←(BAG),
  T2←(BAG ←X),
  T3←(BAG ←Y),
  T4←(BAG)},
 {S1←(BAG ←X),
  S2←(BAG ←Y),
  S3←(BAG),
  S4←(BAG ←X ←Y),
  T1←(BAG),
  T2←(BAG ←X),
  T3←(BAG),
  T4←(BAG ←Y)}}.
```

Unifying the values of S3 with (BAG A), we obtain

```
{{S1←(BAG A),
  S2←(BAG ←Y),
  S3←(BAG A),
  S4←(BAG ←Y),
  T1←(BAG A),
  T2←(BAG),
  T3←(BAG),
  T4←(BAG ←Y),
  X←A},
 {S1←(BAG ←X),
  S2←(BAG A),
  S3←(BAG A),
  S4←(BAG ←X),
  T1←(BAG),
  T2←(BAG ←X),
  T3←(BAG A),
  T4←(BAG),
  Y←A}}.
```

Unifying the values of S4 with (BAG B), we obtain

```
{{S1←(BAG A),
  S2←(BAG B),
  S3←(BAG A),
  S4←(BAG B),
  T1←(BAG A),
  T2←(BAG),
  T3←(BAG),
  T4←(BAG B),
  X←A,
  Y←B},
 {S1←(BAG B),
  S2←(BAG A),
  S3←(BAG A),
  S4←(BAG B),
  T1←(BAG),
  T2←(BAG B),
  T3←(BAG A),
  T4←(BAG),
  Y←A,
  X←B}}
```

or, removing the now useless newly formed fragment variables, {{X←A, Y←B}, {X←B, Y←A}}, just what was expected.

Lemma 3.1. Every non-negative integral solution $(x_1,\ldots,x_m,y_1,\ldots,y_n)$ to the equation $a_1x_1+\ldots+a_mx_m =$

$b_1y_1+\ldots+b_ny_n$ with positive integral coefficients $a_1,\ldots,a_m,b_1,\ldots,b_n$ is an additive linear combination of non-negative integral solutions with value $a_1x_1+\ldots+a_mx_m$ $(=b_1y_1+\ldots+b_ny_n)$ less than or equal to the maximum of m and n times the maximum of the least common multiples of pairs of numbers one from $a_1,\ldots,a_m$ and one from $b_1,\ldots,b_n$.

Proof. Assume with no loss of generality that the least common multiple (lcm) of a_1 and b_1 is the maximum of the least common multiples and that $m \geq n$.

Proof by induction on the value of a solution k.

k=0. The solution with k=0 with $x_1=0, \ldots, x_m=0, y_1=0, \ldots, y_n=0$ is generable as the additive linear combination of non-negative integral solutions with value less than or equal to $m*lcm(a_1,b_1)$ with zero coefficients.

Assume the lemma is true for every non-negative integral solution with value less than or equal to k. Prove it is true for k.

Case 1. $k \leq m*lcm(a_1,b_1)$. In this case, the solution is included among the non-negative integral solutions with value less than or equal to $m*lcm(a_1,b_1)$ and the lemma is true.

Case 2. $k > m*lcm(a_1,b_1)$. Since $a_1x_1+\ldots+a_mx_m = k > m*lcm(a_1,b_1)$, and each $a_ix_i>0$, at least one a_ix_i must be greater than $lcm(a_1,b_1)$, and x_i must be greater than $lcm(a_1,b_1)/a_i$. Similarly, since $b_1y_1+\ldots+b_ny_n = k > m*lcm(a_1,b_1)$, and each $b_jy_j>0$, and $n \leq m$, at least one b_jy_j must be greater than $lcm(a_1,b_1)$, and y_j must be greater than $lcm(a_1,b_1)/b_j$. Consider

the solution with $x_i=\text{lcm}(a_i,b_j)/a_i$, $y_j=\text{lcm}(a_i,b_j)/b_j$, and all other variables zero. This is just the solution in lowest terms involving only x_i and y_j and has value $\text{lcm}(a_i,b_j) \leq \text{lcm}(a_1,b_1)$. Since $\text{lcm}(a_1,b_1)/a_i \geq \text{lcm}(a_i,b_j)/a_i$ and $\text{lcm}(a_1,b_1)/b_j \geq \text{lcm}(a_i,bj)/b_j$ by the maximality of $\text{lcm}(a_1,b_1)$, the second solution involving only x_i and y_j can be subtracted from the first leaving a non-negative integral solution as result. But this difference solution has value $k-\text{lcm}(a_i,b_j)<k$ and is thus composable from solutions with value less than or equal to $m*\text{lcm}(a_1,b_1)$. Therefore, the solution with value $k>m*\text{lcm}(a_1,b_1)$ is the sum of some solution involving only x_i and y_j with value less than or equal to $\text{lcm}(a_1,b_1)$ and some other set of solutions with value less than or equal to $m*\text{lcm}(a_1,b_1)$ and the lemma is true for this case. □

The lemma proves an upper bound on solution values that must be examined in the determination of a complete set of non-negative integral solutions which span the non-negative integral solution space by addition. However, the proven bound is greater than the previously conjectured bound for which we do not have a proof. Although a proof for the tighter bound would be desirable, it should be noted that a lower proven bound does not reduce the number of found solutions theoretically necessary (or, equivalently, the number of introduced fragment variables), but only decreases the cost of computing them. This is true since any additional solutions discovered using a higher bound than necessary must be composable from solutions bounded by any lower proven bound and would therefore be omitted.

It should also be noted that the conjectured bound is a lower bound on solution values which must be examined, i.e., solutions with at least this value need to be examined. This is because one of the needed solutions not otherwise generable is the pairwise solution involving only the variables with those two coefficients with the maximum least common multiple and having value equal to the conjectured bound.

The solutions to examples presented here are complete, and the needed equation solutions were discovered within the lower conjectured bound and no additional (not otherwise composable) solutions were discovered within the higher proven bound.

Theorem 3.7. The unification algorithm for bags composed entirely of fragment variables always terminates, is sound, and is complete.

Proof. The algorithm is guaranteed to terminate since it performs a finite number of operations on the finite number of non-negative integral solutions generated from the equation corresponding to the two bags. The generation of these solutions is finite due to the trial solution values being bounded.

The algorithm is sound since each solution of the derived equation causes the introduction into each of the instantiated bags of an equal number of new fragment variable occurrences. Thus, the two instantiated bags have the same number of occurrences of each new fragment variable and are therefore unified.

Let k be the cardinality of the set of elements of the unified

bags for any solution to the unification of two bags composed entirely of fragment variables. The two instantiated bags must have an equal number of occurrences of each element. That is, $a_1c_{i1}+\ldots+a_mc_{im} = b_1d_{i1}+\ldots+b_nd_{in}$ $(1\leq i\leq k)$ where $a_1,\ldots,a_m,b_1,\ldots,b_n$ are the multiplicities of the fragment variables in the bags as before, c_{ij} is the number of occurrences (in the unifying substitution) of element i in fragment variable j from the first bag, and d_{ij} is the number of occurrences of element i in fragment variable j from the second bag.

Each tuple $(c_{i1},\ldots,c_{im},d_{i1},\ldots,d_{in})$ is a solution to the equation $a_1x_1+\ldots+a_mx_m=b_1y_1+\ldots+b_ny_n$ corresponding to the two bags. It can thus (according to the lemma) be formed as the sum of certain non-negative integral solutions to the equation weighted by positive integers.

Include in the value of the new fragment variable associated with each of these solutions a number of occurrences of element i equal to the coefficient of the solution in the weighted sum. This will result in the assignment of c_{ij} occurrences of element i to each fragment variable j of the first bag and d_{ij} occurrences of element i to each fragment variable j of the second bag.

Do this for each of the k elements in the solution. Let no other or additional elements be included in the values of the new fragment variables. In particular, assign the empty bag to those fragment variables not used above.

This assignment of elements in the solution to new fragment

variables associated with equation solutions generated in the unification process results in the correct number c_{ij} or d_{ij} of each element being assigned to each fragment variable of the original two bags.

Thus, any solution to the unification of two bags is an instance of the returned unifier and the algorithm is complete. □

The final theorem for bag unification relates to its termination, soundness, and completeness in the general case. We cannot prove termination, soundness, and completeness for a completely arbitrary selection of elements of the two bags if such results are not proven for unification of the elements among themselves directly. This is really no restriction on the algorithm's usefulness; no more powerful result could be proved for any bag unification algorithm. For example, we cannot expect to completely unify (BAG (VECTOR A ↔X)) and (BAG (VECTOR ↔X A)) just because the vectors are included in bags presented to a "complete" bag unification algorithm if we cannot completely unify (VECTOR A ↔X) and (VECTOR ↔X A) directly.

What the theorem states is that the algorithm introduces no additional non-termination, unsoundness, or incompleteness. For example, the semi-unification algorithms terminate, are sound, and are complete for expressions containing no fragment variables. Thus, bag unification for bags composed of fragment variables and fragment variable free elements terminates, is sound, and is complete when using semi-unification in the recursive calls to unification algorithms.

Theorem 3.8. The bag unification algorithm always terminates, is sound, and is complete (provided unification of elements is).

Proof. Let a and b be any two bags being unified. Let a^* and b^* be the bags resulting from replacing elements which are not fragment variables by new fragment variables. a^* and b^* are generalizations of a and b respectively, i.e., $a^*\theta=a$ and $b^*\theta=b$ for some θ of the form $\{\ldots,x_i \leftarrow (\text{BAG } y_i),\ldots\}$ where each x_i is a new fragment variable and each y_i is the expression in a or b it replaces in a^* or b^*.

Let σ be the unifier of a^* and b^* returned by the unification algorithm for bags composed entirely of fragment variables. According to the previous theorem, bag unification terminates, is sound, and is complete for bags composed entirely of fragment variables.

Let θ' be θ with substitution σ applied to each variable value of θ. This operation is necessary to cover the possibility that some element of the bags includes an occurrence of a variable also occurring as a fragment variable element of the bags which is thus assigned a value in σ.

$a^*\theta'$ is an instance of a and $b^*\theta'$ is an instance of b. Nevertheless, every unifier of a and b is also a unifier of $a^*\theta'$ and $b^*\theta'$ since σ (by which θ was instantiated to obtain θ') is the most general unifier of generalizations a^* and b^* of a and b.

We now have (by virtue of the instantiation of θ by σ) two substitutions (θ' and σ) for variables of a^* and b^* neither of which includes in variable values any variable being substituted for in either substitution.

Every unifier of a and b is a simultaneous instance of θ' and σ. Thus, all unifiers can be generated by unifying in all possible ways variable values in θ' and σ corresponding to the same variables.

Let $x_1,\ldots,x_k$ be the variables substituted for in θ' and σ and let $y_1,\ldots,y_k$ and $z_1,\ldots,z_k$ be the corresponding values in θ' and σ respectively. (Note that σ contains a substitution pair for every fragment variable in a^* and b^*; θ' contains substitution pairs only for new fragment variables introduced in the formation of a^* and b^*. For variables x_i substituted for in σ but not in θ' we will assume y_i to be x_i, i.e., the null substitution will be used.)

$\{\sigma\sigma_1\ldots\sigma_k \mid \sigma_1 \in \text{unify}[y_1;z_1] \wedge \ldots \wedge \sigma_k \in \text{unify}[y_k\sigma_1\ldots\sigma_{k-1};z_k\sigma_1\ldots\sigma_{k-1}]\}$ is thus the list of all unifiers of a and b. The algorithm thus terminates, is sound, and is complete if each recursive call on unify terminates, is sound, and is complete.

Let us examine the arguments of these recursive calls more closely. Each y_i is a bag consisting of a single element which is not a fragment variable and each z_i is a bag consisting of fragment variables. Thus, every $y_i\sigma_1\ldots\sigma_{i-1}$ is merely an instance of y_i having a single element which is not a fragment variable and every $z_i\sigma_1\ldots\sigma_{i-1}$ is a bag consisting of fragment variables and instances of elements of $z_1,\ldots,z_{i-1}$.

In unifying $y_i\sigma_1\ldots\sigma_{i-1}$ and $z_i\sigma_1\ldots\sigma_{i-1}$, two cases are possible. In the first case, the single element of $y_i\sigma_1\ldots\sigma_{i-1}$

is assigned as the single bag element of a fragment variable of $z_i\sigma_1\ldots\sigma_{i-1}$ and the remainder of $z_i\sigma_1\ldots\sigma_{i-1}$ is unified with the empty bag. This case can be handled with termination, soundness, and completeness by semi-unification. In the second case, the element of $y_i\sigma_1\ldots\sigma_{i-1}$ is unified against an element of $z_i\sigma_1\ldots\sigma_{i-1}$ which is not a fragment variable and is itself an instance of some z_j. Here, the assumption of the theorem is required to complete the argument for total termination, soundness, and completeness of the algorithm.

Thus, the algorithm always terminates, is sound, and is complete provided pairs of elements from the two bags can be unified with termination, soundness, and completeness. □

3.15. AC unification

We present here an algorithm (called the AC unification algorithm) for unifying two terms whose function is associative and commutative. Terms will be represented as if the function had an arbitrary number of arguments with no superfluous parentheses.

We will assume that the argument lists of the two terms being unified have no common arguments. This presents no difficulty since no unifiers are lost and efficiency is gained if common arguments are eliminated immediately. This is done by removing common arguments a pair at a time, one from each of the argument

lists. For example, before unifying f(xxyabc) and f(bbbcz), the b's common to the two terms are removed yielding f(xxyac) and f(bbcz), and the c's common to the two new terms are removed yielding f(xxya) and f(bbz). An example of the utility of immediately removing common arguments is the unification of f(g(x)y) and f(g(x)g(a)). If the g(x)'s common to the two terms are immediately removed, the unification algorithm will return the most general unifier {y←g(a)}. If the common g(x)'s are retained, unification will likely result in the generation of the additional less general unifier {x←a,y←g(a)}.

The algorithm will be expressed partially in terms of an algorithm for the complete unification of terms with an associative and commutative function with only variables as arguments. The result of unifying such terms is an assignment to each variable of the terms some sequence of terms. Each variable is assigned a term t_i (whose function symbol is not f) or a term $f(t_1^{n_1}...t_m^{n_m})$ (with n_i occurrences of term t_i as arguments of f). For such an assignment to be a unifier, the only requirement is that for each term t_i used in any assignment there are the same number of occurrences of that term occurring as arguments of f in each of the unified terms instantiated by the assignment. For example, in unifying $f(x_1x_1x_2x_3)$ and $f(y_1y_1y_2)$, if term t is part of some assignment to one of the variables, then 2 times the number of occurrences of t in the assignment for x_1 plus the number of occurrences of t in the assignment for x_2 plus the number of occurrences of t in the assignment for x_3 must equal 2

times the number of occurrences of t in the assignment for y_1 plus the number of occurrences of t in the assignment for y_2. For example, $\{x_1 \leftarrow f(bb), x_2 \leftarrow f(ab), x_3 \leftarrow a, y_1 \leftarrow b, y_2 \leftarrow f(aabbb)\}$ is a unifier of $f(x_1x_1x_2x_3)$ and $f(y_1y_1y_2)$ since there are 2 a's and 5 b's in the instantiations of $f(x_1x_1x_2x_3)$ and $f(y_1y_1y_2)$ with the unified term being f(aabbbbb).

With each pair of terms with an associative and commutative function with only variable arguments is associated a single equation representing the number and multiplicity of variables in each term. For example, the equation $2x_1+x_2+x_3 = 2y_1+y_2$ is associated with the pair of terms given above. This equation succintly represents the condition for a substitution to be a unifier: that the sum of the number of occurrences of any term in the value of each variable multiplied by the multiplicity of the variable in the term must be equal for the two terms.

Non-negative integral solutions to such equations can be used to represent unifiers. The solutions must be non-negative integral since each variable must be assigned a non-negative integral number of occurrences of each term.

In order to generate all the solutions to the problem of unifying the two terms, it is necessary to be able to represent all the solutions to the equation derived from the terms. Every non-negative integral solution to the equation is representable as a sum of elements of a particular finite set of non-negative integral solutions to the equation, i.e., every non-negative integral solution to the equation is a sum (equivalently, a sum

with non-negative integral weights) of elements of a particular finite set of non-negative integral solutions. The finite set of non-negative integral solutions by whose addition the entire non-negative integral solution space is spanned is generable by generating in ascending order of value solutions to the equation, eliminating those solutions composable from those previously generated. This process can be made finite by placing a bound on the maximum solution value which will be used; such a maximum was proved in Lemma 3.1 to eliminate no needed solutions.

Consider the equation $2x_1+x_2+x_3 = 2y_1+y_2$. Solutions to the equation are:

	x_1	x_2	x_3	y_1	y_2	$2x_1 \pm x_2 \pm x_3$	$2y_1 \pm y_2$	
1	0	0	1	0	1	1	1	z_1
2	0	1	0	0	1	1	1	z_2
3	0	0	2	1	0	2	2	z_3
4	0	1	1	1	0	2	2	z_4
5	0	2	0	1	0	2	2	z_5
6	1	0	0	0	2	2	2	z_6
7	1	0	0	1	0	2	2	z_7

Associated with each solution above is a new variable (in the rightmost column). The assignment of as many occurrences of that variable as specified in the solution to each of the variables of the original term results in a partial solution to the unification of the the original terms. In particular, the assignment of 2 occurrences of variable z_3 to x_3 and 1 occurrence

to y_1 results in an equal number of occurrences of variable z_3 in each of $f(x_1x_1x_2x_3)$ and $f(y_1y_2)$.

Every non-negative integral solution to the equation is a (non-negative integer weighted) sum of the 7 solutions presented above, i.e., every solution is representable as $x_1=z_6+z_7$, $x_2=z_2+z_4+2z_5$, $x_3=z_1+2z_3+z_4$, $y_1=z_3+z_4+z_5+z_7$, $y_2=z_1+z_2+2z_6$ for some non-negative integral values of $z_1,\ldots,z_6$. However, not every solution to the equation is a solution to the unification problem for which the equation was derived. There is an additional constraint that each variable of the original terms must be have at least one term in its value; it cannot have zero terms in its value.

Hence, we must form that subset of the $2^7=128$ sums for which each element of the 5-tuple is non-zero. (It is not necessary to consider sums in which any solution has a coefficient other than 0 or 1 since such solutions (in the unification problem) are already representable since the solution's inclusion with coefficient 1 introduces a variable which can have as its value an arbitrary number of terms as arguments of f thus simulating the case of the coefficient being greater than 1.) There are 69 such sums including for example (representing the sum by the set of its indices) {2,3,6}, {1,2,3,6}, and {4,6} with associated unifiers

$\{x_1 \leftarrow z_6,\ x_2 \leftarrow z_2,\ x_3 \leftarrow f(z_3z_3),\ y_1 \leftarrow z_3,\ y_2 \leftarrow f(z_2z_6z_6)\}$,

$\{x_1 \leftarrow z_6,\ x_2 \leftarrow z_2,\ x_3 \leftarrow f(z_1z_3z_3),\ y_1 \leftarrow z_3,\ y_2 \leftarrow f(z_1z_2z_6z_6)\}$, and

$\{x_1 \leftarrow z_6,\ x_2 \leftarrow z_4,\ x_3 \leftarrow z_4,\ y_1 \leftarrow z_4,\ y_2 \leftarrow f(z_6z_6)\}$.

Note that if a variable could have as its value zero terms rather than one or more terms as in the first order predicate calculus, it would be unnecessary to form this subset of 2^n (where n is the number of solutions) sums. Only the sum of all the solutions would be required since any variable present in this sum could have value zero, and the variables in the corresponding unifier could be matched against zero terms. This is the situation with fragment variables in the bag data type in QA4 and QLISP [43,37].

To be more precise in the definition of the algorithm, the algorithm consists of the following steps:

1. Form an equation from the two terms where the coefficient of each variable in the equation is equal to the multiplicity of the corresponding variable in the term.
2. Generate all non-negative integral solutions to the equation eliminating all those solutions composable from other solutions.
3. Associate with each solution a new variable.
4. For each sum of the solutions (no solution occurring in the sum more than once) with no zero components assemble a unifier composed of assignments to the original variables with as many of each new variable as specified by the solution element in the sum associated with the new variable and the original variable.

Now we present the complete algorithm for unifying general terms with associative and commutative functions using the algorithm for the variable only case above. We are here concerned with terms whose function is associative and

commutative with arbitrary arguments, i.e., arguments that may contain ordinary (non-associative, non-commutative) functions or f or other functions which are associative and commutative. We assume the presence of ordinary unification to deal with those aspects of the unification problem not dealt with explicitly here.

First, when unifying two terms, two new terms with only variable arguments are formed by uniformly replacing distinct arguments by new variables. These new terms have only variable arguments and are generalizations of the original two terms. For example, in unifying f(xxya) and f(bbz), we form generalizations $f(x_1x_1x_2x_3)$ and $f(y_1y_1y_2)$ with substitution $\{x_1 \leftarrow x, x_2 \leftarrow y, x_3 \leftarrow a, y_1 \leftarrow b, y_2 \leftarrow z\}$ instantiating the new terms to the original terms.

Next, using the previous algorithm for the variable only case, we unify the generalizations of the original terms. This has already been done for the example above resulting in 69 unifiers as stated previously.

Now we have the generalizations of the two original terms, a substitution to instantiate them to the original terms, and a complete set of their unifiers. Every unifier of the original terms is a simultaneous instance of the substitution to instantiate the generalizations to the original terms and a unifier of the generalizations. So all that is necessary to get all the unifiers of the original terms is to unify (for each variable being substituted for) the value in the substitution and the value in the unifiers.

In the example, x_3 must have value a and y_1 must have value b. Thus, any unifier of $f(x_1x_1x_2x_3)$ and $f(y_1y_1y_2)$ which assigns to x_3 or y_2 a non-variable, i.e., a term of the form f(...) may be immediately excluded from consideration since the unification of it with the assignment including $x_3 \leftarrow a$ and $y_1 \leftarrow b$ will fail. (This constraint could be applied during the generation of sums of solutions to the equation rather than afterwards.) This constraint eliminates 63 of the 69 unifiers, leaving sums (1) {4,6}, (2) {2,4,6}, (3) {1,5,6}, (4) {1,2,5,6}, (5) {1,2,7}, and (6) {1,2,6,7} with associated unifiers

(1) $\{x_1 \leftarrow z_6, x_2 \leftarrow z_4, x_3 \leftarrow z_4, y_1 \leftarrow z_4, y_2 \leftarrow f(z_6z_6)\}$,

(2) $\{x_1 \leftarrow z_6, x_2 \leftarrow f(z_2z_4), x_3 \leftarrow z_4, y_1 \leftarrow z_4, y_2 \leftarrow f(z_2z_6z_6)\}$,

(3) $\{x_1 \leftarrow z_6, x_2 \leftarrow f(z_5z_5), x_3 \leftarrow z_1, y_1 \leftarrow z_5, y_2 \leftarrow f(z_1z_6z_6)\}$,

(4) $\{x_1 \leftarrow z_6, x_2 \leftarrow f(z_2z_5z_5), x_3 \leftarrow z_1, y_1 \leftarrow z_5, y_2 \leftarrow f(z_1z_2z_6z_6)\}$,

(5) $\{x_1 \leftarrow z_7, x_2 \leftarrow z_2, x_3 \leftarrow z_1, y_1 \leftarrow z_7, y_2 \leftarrow f(z_1z_2)\}$, and

(6) $\{x_1 \leftarrow f(z_6z_7), x_2 \leftarrow z_2, x_3 \leftarrow z_1, y_1 \leftarrow z_7, y_2 \leftarrow f(z_1z_2z_6z_6)\}$.

Unifying each of these with $\{x_1 \leftarrow x, x_2 \leftarrow y, x_3 \leftarrow a, y_1 \leftarrow b, y_2 \leftarrow z\}$, we obtain

(1) no unifier since $z_4 \leftarrow a$ and $z_4 \leftarrow b$ are not unifiable,

(2) no unifier since $z_4 \leftarrow a$ and $z_4 \leftarrow b$ are not unifiable,

(3) $\{x \leftarrow z_6, y \leftarrow f(bb), z \leftarrow f(az_6z_6)\}$ (= $\{y \leftarrow f(bb), z \leftarrow f(axx)\}$),

(4) $\{x \leftarrow z_6, y \leftarrow f(bbz_2), z \leftarrow f(az_2z_6z_6)\}$ (= $\{y \leftarrow f(bbz_2), z \leftarrow f(az_2xx)\}$),

(5) $\{x \leftarrow b, y \leftarrow z_2, z \leftarrow f(az_2)\}$ (= $\{x \leftarrow b, z \leftarrow f(ay)\}$), and

(6) $\{x \leftarrow f(bz_6), y \leftarrow z_2, z \leftarrow f(az_2z_6z_6)\}$ (= $\{x \leftarrow f(bz_6), z \leftarrow f(ayz_6z_6)\}$).

This is a complete set of unifiers of f(xxya) and f(bbz).

Since x_3 and y_1 of the variable only case correspond to a and

b respectively, and a and b are not unifiable, any sum including solution 4 to the equation $2x_1+x_2+x_3 = 2y_1+y_2$ can be excluded from consideration since it would require (as in (1) and (2) above) the unification of a and b. As with the constraint on variables corresponding to non-variable terms not being assigned more than one variable (terms of the form f(...)) in the variable only case, this latter constraint on solutions can be applied during the generation of unifiers in the variable only case rather than afterwards. Elimination of solution 4 before generation of the 2^n sums, and elimination of sums which do not meet the first constraint would result in the formation only of unifiers (3), (4), (5), and (6) of the variable only case, each of which has a corresponding unifier in the general case.

More precisely, the algorithm consists of the following steps:

1. Form generalizations of the two terms replacing each distinct argument by a new variable.
2. Use the algorithm for the variable only case to generate unifiers for the generalizations of the two terms. The variable only case algorithm may be constrained to eliminate the generation of unifiers assigning more than one term to variables whose value must be a single term, and the generation of unifiers which will require the later unification of terms which are obviously not unifiable.
3. Unify for each variable in the substitution from step 1 and the unifiers from step 2 the variable values and return the resulting assignments for variables of the original terms. This is a complete set of unifiers of the original terms.

We will first establish the validity of eliminating arguments common to the two terms. This will be done by proving that any unifier of the terms is a unifier of the terms with a pair of common arguments removed and vice versa.

Theorem 3.9. Let $s_1,\ldots,s_m,t_1,\ldots,t_n$ be terms with $s_i=t_j$ for some i,j. Let θ be a unifier of $f(s_1\ldots s_m)$ and $f(t_1\ldots t_n)$, and let σ be a unifier of $f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)$ and $f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)$. Then (1) θ is a unifier of $f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)$ and $f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)$, and (2) σ is a unifier of $f(s_1\ldots s_m)$ and $f(t_1\ldots t_n)$.

Proof.

1. $f(s_i\theta f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)\theta) = f(s_1\ldots s_m)\theta = f(t_1\ldots t_n)\theta = f(t_j\theta f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)\theta)$, and $s_i\theta=t_j\theta$. Therefore $f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)\theta = f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)\theta$ and θ is a unifier of $f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)$ and $f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)$.

2. $f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)\sigma = f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)\sigma$ and $s_i\sigma=t_j\sigma$. Therefore $f(s_i\sigma f(s_1\ldots s_{i-1}s_{i+1}\ldots s_m)\sigma) = f(s_1\ldots s_m)\sigma = f(t_1\ldots t_n)\sigma = f(t_j\sigma f(t_1\ldots t_{j-1}t_{j+1}\ldots t_n)\sigma)$ and σ is a unifier of $f(s_1\ldots s_m)$ and $f(t_1\ldots t_n)$. □

Theorem 3.10. The AC unification algorithm for terms with associative and commutative function with only variables as arguments always terminates, is sound (returns no substitutions which are not unifiers), and is complete (every unifier is an instance of a returned unifier).

Proof. The algorithm is guaranteed to terminate since it

performs a finite number of operations on the finite number of non-negative integral solutions generated from the equation corresponding to the two terms. The generation of these solutions is finite due to the trial solution values being bounded.

The algorithm is sound since each solution of the derived equation causes the introduction into each of the instantiated terms of an equal number of new variable occurrences. Thus, the two instantiated terms have the same number of occurrences of each new variable and are therefore unified.

Any unifier must assign to each variable a term of the form t_i (whose function symbol is not f) or a term $f(t_1^{n_1}...t_m^{n_m})$ (with n_i occurrences of term t_i as arguments of f). Let k be the cardinality of the set of such terms t_i in any solution to the unification of a pair of terms with only variables as arguments. The two instantiated terms must have an equal number of occurrences of each of these k terms as arguments of f. That is, $a_1c_{i1}+...+a_mc_{im} = b_1d_{i1}+...+b_nd_{in}$ $(1 \leq i \leq k)$ where m is the number of distinct variables in the first term being unified, n is the number of distinct variables in the second term, a_j is the multiplicity of the j^{th} variable in the first term, b_j is the multiplicity of the j^{th} variable in the second term, c_{ij} is the number of occurrences of term i in variable j in the first term, and d_{ij} is the number of occurrences of term i in variable j in the second term.

Each tuple $(c_{i1},...,c_{im},d_{i1},...,d_{in})$ is a solution to the

equation $a_1x_1+\ldots+a_mx_m = b_1y_1+\ldots+b_ny_n$ corresponding to the terms being unified. It can thus (according to Lemma 3.1) be formed as the sum of certain non-negative integral solutions to the equation weighted by positive integers.

Consider the unifier corresponding to the sum of all those solutions to the equation which are required in the formation of any of the tuples $(c_{il},\ldots,c_{im},d_{il},\ldots,d_{in})$. We will show that the hypothesized unifier is an instance of this unifier returned by the algorithm.

Include in the value of the new variable associated with each of these solutions a number of occurrences of term i equal to the coefficient of the solution in the weighted sum. This will result in the proper assignment of c_{ij} occurrences of term i to each variable j of the first term and d_{ij} occurrences of term i to each variable j of the second term.

Do this for each of the k terms in the solution. Let no other or additional terms be included in the values of the new variables.

This assignment of terms in the solution to new variables associated with equation solutions generated in the unification process results in the correct number c_{ij} or d_{ij} of each term being assigned to each variable of the original two terms.

Thus, any solution to the unification of two terms with only variables as arguments is an instance of a returned unifier and the algorithm is complete. □

Theorem 3.11. The AC unification algorithm for general terms

with associative and commutative function always terminates, is sound, and is complete.

Proof. Let s and t be any two terms being unified. Let s^* and t^* be the terms resulting from replacing each distinct term by a new variable. s^* and t^* are generalizations of s and t respectively, i.e., $s^*\theta=s$ and $t^*\theta=t$ for some θ of the form $\{\ldots,x_i \leftarrow c_i,\ldots\}$ where each x_i is a new variable and each c_i is the term in s or t it replaces in s^* or t^*.

Let $\{\sigma_j\}$ denote the unifiers of s^* and t^* returned by the unification algorithm for terms with associative and commutative function with only variables as arguments. Each σ_j is of the form $\{\ldots,x_i \leftarrow d_i,\ldots\}$ where each x_i is a variable of s^* or t^* and d_i is the term assigned to it by the unification algorithm. According to the previous theorem, unification terminates, is sound, and is complete for this case.

Simultaneous instances of θ and σ_j represent unifiers of s and t since $s^*\theta=s$, $t^*\theta=t$, and $s^*\sigma_j=t^*\sigma_j$.

Unifying each c_i with each d_i of a returned unifier σ_j of s^* and t^* results in (by the assumption of termination, soundness, and completeness of the recursive call on the unification algorithm for terms of lesser complexity) a complete set of unifiers for the original terms s and t. □

We have presented an algorithm for unifying general terms with associative and commutative function. We have proven that the algorithm is guaranteed to terminate, is sound, and is complete.

The advantages of this algorithm as compared to other

approaches to unifying such terms are that the associativity and commutativity properties need not be axiomatized and that all the unifiers of a pair of such terms are immediately returned eliminating the unnecessary and redundant computation often occurring in other approaches which generate only some of the unifiers at each step with no indication of when all the unifiers have been generated.

3.16. Class unification

It is possible to write a complete class unification algorithm in the style of the previously presented non-terminating vector unification algorithm. The algorithm presented below is sound and complete though non-terminating just as the vector unification algorithm was. However, adding extra constraints on substitutions for fragment variables (not described in the algorithm proper, but presented later) eliminates this difficulty, resulting in an algorithm which is sound, complete, and guaranteed to terminate.

Since class unification requires an element of one class possibly to be matched against more than one element of another class, matched elements cannot be simply removed and disregarded. Previously matched elements must be retained. They may be kept in an additional variable in the algorithm or, for simplicity of description and implementation, be retained in the original classes themselves as in the algorithm being presented here. The

following conventions are followed by the algorithm: previously matched elements of two classes are retained in the right end (tail) of the two classes and the leftmost of any two duplicate elements in a single class is immediately eliminated. Two classes are fully unified when they are identical (all the elements are previously matched elements in the common tail of the two classes).

The unifyclass procedure, when presented with two non-empty classes with no previously unmatched common elements or duplicate elements within a single class, always matches the first element of the first class at each step after a possible interchange of the two classes. The interchange is a normalization operation whose result is that the first element of the first class after the interchange is guaranteed to be previously unmatched.

Class unification proceeds almost exactly as does class semi-unification described earlier except only a single element of the opposing class is included in each substitution for a fragment variable as in vector unification described earlier.

unifyclass[x; y]=

1. If x is of the form (CLASS α a γ a ϵ) for arbitrary element a then unify[(CLASS α γ a ϵ); y].
2. If y is of the form (CLASS β a δ a ϵ) for arbitrary element a then unify[x; (CLASS β δ a ϵ)].
3. If x is of the form (CLASS α a γ) and y is of the form (CLASS β a δ) for arbitrary element a and $\gamma \neq \delta$ then unify[(CLASS α γ a); (CLASS β δ a)].

4. If x is of the form (CLASS α) and y is of the form (CLASS β α) then unify[y; x].

5. If x is of the form (CLASS $\leftrightarrow$X α) then

   a. for each fragment variable $\leftrightarrow$Y in y (y possibly reordered is of the form (CLASS $\leftrightarrow$Y β)) let θ={X$\leftarrow$(CLASS $\leftrightarrow$T $\leftrightarrow$X', Y$\leftarrow$(CLASS $\leftrightarrow$T $\leftrightarrow$Y')} and return {$\theta\sigma$ | $\sigma \in$ unify[xθ; yθ]} and

   b. for each element b in y which is not a fragment variable and does not contain an instance of variable X (y possibly reordered is of the form (CLASS b β)) let θ={X$\leftarrow$(CLASS b $\leftrightarrow$X')} and return {$\theta\sigma$ | $\sigma \in$ unify[xθ; yθ]} and

   c. let θ={X$\leftarrow$(CLASS)} and return {$\theta\sigma$ |$\sigma \in$ unify[xθ; yθ]}.

6. If x is of the form (CLASS a α) where element a is not a fragment variable then

   a. for each fragment variable $\leftrightarrow$Y in y for which a does not contain an instance of variable Y (y possibly reordered is of the form (CLASS $\leftrightarrow$Y β)) let θ={Y$\leftarrow$(CLASS a $\leftrightarrow$Y')} and return {$\theta\sigma$ | $\sigma \in$ unify[xθ; yθ]} and

   b. for each element b in y which is not a fragment variable (y possibly reordered is of the form (CLASS b β)) return {$\theta\sigma$ | $\theta \in$ unify[a; b] $\wedge$ $\sigma \in$ unify[xθ; yθ]}.

Notes:

1. If a class has a repeated element, the first occurrences of that element are deleted (steps 1, 2).

2. If two elements of two classes not in the common tail of the classes (not previously matched) are identical, the classes with the elements moved to the end are unified (step 3).

3. To insure that the next substitution operation use a previously unmatched element, the classes are interchanged if the second is a tail of the first (step 4).

4. If the first element of x is a fragment variable, each substitution composed of a non-fragment variable element of y not containing the variable being substituted for and a new fragment variable is performed and the instantiated classes are unified for each substitution (step 5b). This forms a most general substitution for the fragment variable including each opposed non-fragment variable element. For each fragment variable element of y, the fragment variable of y is split into two fragment variables and one is included in a substitution for the first element of x as above and the instantiated classes are unified for each substitution (step 5a). This forms a most general substitution for the two fragment variables expressing their common (↔T) and disjoint (↔X', ↔Y') parts. In addition, the empty class is substituted for the first element of x (step 5c).

5. If the first element of x is not a fragment variable, it is unified with each non-fragment variable element of y and the classes instantiated by the returned unifiers are unified (step 6b). For each fragment variable element of y not contained in the first element of x, the substitution for the fragment variable of the first element of x and a new fragment variable is performed and the instantiated classes are unified for each substitution (step 6a). This forms a most general substitution for the fragment variable including the first element of x.

The algorithm can readily be shown to be sound using a proof similar to the proof for (class) semi-unification. Steps 1, 2, 3, and 4 are the same as for semi-unification and hence their soundness can be justified in the same way. Steps 5 and 6 always perform a substitution on the arguments and recursively call the unification algorithm with the instantiated arguments and return the compositions of the substitution and the returned unifiers. This is also clearly sound.

Completeness is also easy to prove. Steps 1 and 2 merely remove redundant duplicate elements from a single class forming an equivalent class. Step 3 matches identical elements from the two classes. Every previously unmatched fragment variable is permitted to contain any element of the opposed expression. Any previously unmatched non-fragment variable is permitted to be matched to any element of the opposed expression or be contained in any fragment variable of the opposed expression. Any pair of fragment variables can be matched, selecting out their common and disjoint subclasses. The procedure is clearly exhaustive, exploring all alternatives for each previously unmatched element, and forming most general substitutions for each case.

The procedure will not, in general, terminate for the same reasons the vector unification algorithm does not terminate. The algorithm when matching with a fragment variable often maintains the same number of variables in the instantiated expressions to be unified or even increases them. Each of these new fragment variables can include the value of an element of the opposed expression.

For example, consider the unification of (CLASS ↔X) and (CLASS A).

```
unify[(CLASS ↔X); (CLASS A)]
| unify[(CLASS ↔X1 A); (CLASS A)]
| | unify[(CLASS A); (CLASS A)]
| | return {X←(CLASS A), X1←(CLASS)}
| | unify[(CLASS ↔X2 A); (CLASS A)]
| | | unify[(CLASS A); (CLASS A)]
| | | return {X←(CLASS A A), X1←(CLASS A), X2←(CLASS)}
| | | unify[(CLASS ↔X3 A); (CLASS A)]

...
```

The important thing to recognize in the example is that all the returned unifiers are equivalent; repetitions in a class are totally redundant.

For this reason, we can place additional restrictions on the matches performed with fragment variables and still retain completeness. This set of restrictions is adequate to assure termination in all cases.

1. Never perform step 5a on two variables whose ancestors have been matched against each other. Assume $\leftrightarrow X$ is a descendant of $\leftrightarrow X_0$ and $\leftrightarrow Y$ is a descendant of $\leftrightarrow Y_0$, i.e., $\leftrightarrow X$ is $\leftrightarrow X_m$ generated after a sequence of substitutions $\{X_i \leftarrow (\text{CLASS } b_i \leftrightarrow X_{i+1})\}$ are performed and $\leftrightarrow Y$ is $\leftrightarrow Y_n$ generated after a sequence of substitutions $\{Y_j \leftarrow (\text{CLASS } a_j \leftrightarrow Y_{j+1})\}$, and X_0 and Y_0 were matched

against each other with result {X_0←(CLASS ↔T ↔X'), Y_0←(CLASS ↔T ↔Y')}. Then X_m and Y_n should not be unified against each other. X is either descended from X' or from T; Y is either descended from Y' or from T. If X is descended from X' and Y is descended from Y', we can justify not matching X and Y by the assumption that T contained all the common elements of ancestors X_0 and Y_0, and hence X' and Y', and X and Y have no common elements. If X is descended from T and Y is descended from Y', we can justify not matching X and Y since T and Y' together compose Y_0 and thus can be assumed to have no common elements, and hence X and Y have no common elements. If X is descended from X' and Y is descended from T, we can justify not matching X and Y since T and X' together compose X_0 and thus can be assumed to have no common elements, and hence X and Y have no common elements. If both X and Y are descended from T, we can justify not matching X and Y since there is no value to matching elements of the same class against each other directly.

2. Never perform step 5a or step 6a where the element to be included in the value of the fragment variable is already in the value of any ancestor of the fragment variable. Performing this step under these circumstances would result in placing a duplicate element in the value of the ancestor fragment variable. This restriction can be applied when the element is included in the value of any ancestor since, in the case where the fragment variable has multiple ancestors, some containing the element and some not containing it, an alternative sequence of substitutions

would include the element in (a descendant of) a fragment variable representing the common elements of the ancestors.

Since there is a limit on the number of matched elements, being the number of non-fragment variable elements in the original classes (this can readily be seen since each step in the algorithm unifies the same expressions after instantiation or removal of duplicate elements or matching of identical elements), steps 5a and 6a can be performed only a finite number of times, their being a limit on the number of elements that can be included in the values of the original fragment variables which are ancestors to all derived fragment variables.

The introduction of additional fragment variables in step 5b creates no additional difficulty since no descendants of the introduced fragment variables are permitted to be matched against each other.

These two cases account for all the non-terminating behavior of the algorithm, and thus the algorithm with these constraints always terminates, is sound, and is complete.

3.17. Constraints

Another facility provided by QA4 pattern matching is constraints. Two types of constraints can be applied.

The first is a type constraint applied to variables which dictates the type of expression (vector, bag, or class) which must be matched to the variable. For example, +X/BAG represents the requirement that variable X must be bound to a bag.

The type constraint may be readily implemented within the confines of the unification algorithms already provided by replacing (on input or when the expressions are unified) expressions of the form ←var/type by expressions of the form (type ←←var). For example, ←X/BAG would be replaced by (BAG ←←X).

The second is a general predicate constraint which can be applied to any expression (although its effect really varies only with the way variables of the expression are instantiated by the match). For example, (^ (VECTOR 1 ←X) (LT $X 2)) causes a match to fail if the X of (VECTOR ←X) has value greater than or equal to 2. Thus, the expression will match (VECTOR 1 0) but not (VECTOR 1 2).

Given (the conventional pattern matching) assumption that the opposing (target) expression contains no variables, general predicate constraints are easily effected by merely applying the predicate to the opposing expression to determine the truth of the predicate. If the predicate is true, matching continues. If it is false, the current partial match is rejected.

This mechanism does not work in the case of the opposing expression containing variables or other complex pattern matching actors. It is unreasonable to expect (at least in all cases) that the predicate has sufficient knowledge to either propose or determine the existence of instances of variables or pattern matching actors in the opposing expression for which the predicate is true.

Although the pattern matching facilities of QA4/QLISP (disregarding for the moment the added data types of BAGs and CLASSes and considering only VECTORs) appears on superficial examination to be less powerful than the pattern matching language MATCHLESS of PLANNER, the ability to specify general predicate constraints means that any MATCHLESS pattern has a corresponding QA4/QLISP pattern, obtained by merely incorporating all the matching requirement of the MATCHLESS pattern into a general predicate constraint.

There are three principal solutions to the handling of these predicate constraints. The first solution which is logically sound and complete is the use of semi-unification and elaboration procedures so that the constraint checking is applied only to ground expressions, or expressions at some level of complexity fully understood by the constraint expression.

The more pragmatically reasonable second approach is to evaluate the constraint expression even if the variables it is checking are not fully instantiated. It is then the constraint expressions responsibility to decide whether the match should be accepted or not, the decision not necessarily being correct given the lack of full instantiation of the variables.

A third approach would be to use the constrained resolution approach of passing down the constraint and determining its correctness only when X becomes fully instantiated or at the end of the deduction in cases where the variables are not fully instantiated. Thus, if the match otherwise succeeds, and X is

not fully instantiated, the constraint (LT $X 2) would be attached to X and be evaluated as soon as X is fully instantiated, or at the end of the deduction if X isn't fully instantiated before then.

3.18. Algebraic Unification

So far in discussing unification, we have not concerned ourselves at all with the semantics of the symbols appearing in the expressions to be unified, with the exceptions of data types (VECTOR, BAG, CLASS) and, of course, variables. This omission is not because such concerns are unimportant, but because (1) the semantic free unification we have been discussing is important and (2) consideration of the semantics of the symbols often readily leads to areas of great complexity and undecidability of unifiability.

Some results are available for certain interpretations of symbols. Certain theories (for example, group theory) can be formulated in terms of complete sets of reductions (see the next section for a brief discussion of them) These reductions can be applied to a pair of expression, the expressions being equivalent if and only if they reduce to the same expression. Included in this is, of course, handling of the group identity element, which can be simplified out of any expression.

Powerful and useful though complete sets of reductions are, they are not universally applicable, most importantly not being applicable in commutative theories.

Another aspect of algebraic unification is demonstrated by the following example. If we wish to unify the expressions (+ ←X ←Y) and (+ 3 C) where + represents the addition function and is DEFTYPEd BAG, bag unification will return the unifiers {X←3, Y←C} and {X←C, Y←3}. This is the best than any unification algorithm can do without knowing the semantics of +. If we do use what we know about +, then we can return a unifier like {X←(- (+ 3 C) ←Y)}.

The general approach is then to form an equation of the two expressions being unified, e.g., (+ ←X ←Y)=(+ 3 C) and then simplify the equation to provide assignments to some or all of the variables of the equation, using whatever knowledge one has of the semantics of the terms. This can and should incorporate simplifications such as replacing operations entirely on numbers by their results when such applications are formed in rearrangement of terms.

Little can be said concerning general techniques involved in this algebraic unification since it is so thoroughly conditioned on the semantics of the terms involved. In the example above, using * (the multiplication function) instead of + invalidates the unifier {X←(/ (* 3 C) ←Y) because of the singularity of the division function at 0. Sandewall [60] proposes a system for conversion of logical expressions to equivalent procedural forms which includes some facilities for specification of the unification mechanism to be used in invoking one of these procedures.

Some form of algebraic unification is probably essential in many theorem prover based systems such as program verifiers in which the semantics of operations like + must be understood. However, a general need for algebraic unification does not eliminate the utility of the algorithms presented here for unifying such items as vectors, bags, and classes, since such more primitive procedures must be relied upon when nothing but the data type is known about the semantics of the operation.

3.19. Complete sets of reductions

This section is intended to point out the relevance of this current investigation into unification algorithms to a new field of research in mechanical theorem proving. This new area of complete sets of reductions looks very promising though it presently has some significant limitations one of which may be partially addressed by some of the ideas presented in this chapter. This section is rather indefinite and inconclusive, but we hope it demonstrates a wider applicability of the ideas of this chapter than is immediately obvious.

Complete sets of reductions are sets of simplifying rewriting rules the successive applications of which to any term by some algorithm is guaranteed to terminate (the terms "simplifying" and "reduction" refer to this finite termination property) and is guaranteed to terminate uniquely regardless of the order of applications of reductions (the term "complete" refers to this unique termination property).

Complete sets of reductions have been used or suggested in previous systems such as Nevin's theorem prover [34], Guard et al's theorem prover [57], and simplifiers by Slagle [46]. They were first thoroughly formalized by Knuth and Bendix [24] who, most importantly, provide an algoritnm (the Knuth-Bendix extension algorith) which often succeeds in extending an incomplete set of reductions to a complete set. They have also been extensively examined by Lankford [26,27].

Unification is relevant to complete sets of reductions since unification is employed in the test for completeness (unique termination) of a set of reductions. A subterm of the left (unreduced) side of a reduction can be unified with the left side of another to represent the case of both reductions being simultaneously applicable. If in every case where such a unification is possible the result of applying the first production and the result of applying the second reduction simplifies to the same term, the set of reductions is complete.

This test for completeness plays a vital role in the Knuth-Bendix extension algorithm since if in trying to demonstrate completeness of the current set of reductions two different non-reducible terms are generated from the same term the two terms are composed into a new reduction which is proposed as an addition to the current set of reductions.

An example of the performance of Knuth-Bendix extension algorithm (transcribed from a version we wrote in LISP using unification and substitution functions of PSTP) is the following

production of a complete set of reductions for free groups. The algorithm is presented with reductions 1, 2, and 3. These are the left identity axiom and the left inverse axiom and the associative axiom expressed as rewrites. In the case of associativity, "simplicity" of terms is defined such that (faf(bc)) is simpler than f(f(ab)c).

```
(1 (F (E) X) X)
(2 (F (G X) X) X)
(3 (F (F X Y) Z) (F X (F Y Z)))
(4 (F (G X2) (F X2 X1)) X1) FROM 3 AND 2
(5 (F (G (E)) X1) X1) FROM 4 AND 1
(6 (F (G (G (E))) X1) X1) FROM 5 AND 4
(7 (G (E)) (E)) FROM 6 AND 2
6 DELETED
5 DELETED
(8 (F (G (G X1)) (E)) X1) FROM 4 AND 2
(9 (F (G (G X1)) X2) (F X1 X2)) FROM 8 AND 3
8 DELETED
(10 (F X1 (E)) X1)
(11 (G (G X1)) X1) FROM 10 AND 9
9 DELETED
(12 (F X1 (G X1)) (E)) FROM 11 AND 2
(13 (F X1 (F X2 (G (F X1 X2)))) (E)) FROM 12 AND 3
(14 (F X2 (F (G X2) X1)) X1) FROM 12 AND 3
(15 (F X2 (F X3 (F (G (F X2 X3)) X1))) X1) FROM 14 AND 3
(16 (F (G (F X1 X2)) (F X1 (F X2 X3))) X3) FROM 4 AND 3
(17 (F X1 (F X2 (F X3 (G (F X1 (F X2 X3)))))) (E)) FROM 13 AND 3
(18 (G (F (G (F X1 X2)) X1)) X2) FROM 16 AND 2
(19 (F (G (F X1 (G X2))) X1) X2) FROM 16 AND 2
(20 (F (G (F X1 X2)) X1) (G X2)) FROM 19 AND 11
19 DELETED
18 DELETED
(21 (F (G (F X3 X1)) (F X3 X2)) (F (G X1) X2)) FROM 20 AND 3
16 DELETED
(22 (G (F (G X1) X2)) (F (G X2) X1)) FROM 21 AND 2
(23 (G (F X1 X2)) (F (G X2) (G X1))) FROM 22 AND 11
12 DELETED
21 DELETED
20 DELETED
17 DELETED
15 DELETED
13 DELETED
```

Note that the preceding successful completion of the original set of reductions used a complexity ordering in attempting to

complete the set. A depth first ordering goes down a fruitless infinite branch, and a breadth first ordering is likely terminate tne algorithm with failure by producing something like (F (G (F X1 X2)) X1) and (F (G (F X3 X2)) X3) as the final results of simplifying a single term, neither of which is reducible to the other, causing termination of the algorithm with failure.

The final complete set of reductions (written in more conventional form) is:

1. $e*x \rightarrow x$
2. $x^{-1}*x \rightarrow e$
3. $(x*y)*z \rightarrow x*(y*z)$
4. $x^{-1}*(x*y) \rightarrow y$
7. $e^{-1} \rightarrow e$
10. $x*e \rightarrow x$
11. $(x^{-1})^{-1} \rightarrow x$
12. $x*x^{-1} \rightarrow e$
14. $x*(x^{-1}*y \rightarrow y$
23. $(x*y)^{-1} \rightarrow y^{-1}*x^{-1}$

We have been extensively concerned with special representation and unification of associativity and commutativity. This is highly relevant to this new field of complete sets of reductions.

We have adopted a representation for associative functions which permits an arbitrary number of arguments rather than requiring parenthesization. Our unparenthesized form is a canonical form for associativity; likewise, the "all parentheses to the right" form used by present complete sets of reductions technology is a canonical form for associativity.

We feel that the use of the "all parentheses to the right" form has certain dire consequences for the size and complexity of the complete set of reductions when compared with the unparenthesized form.

If the unparenthesized form is used, reduction 3 can of course be eliminated, and reductions 4 and 14 can be eliminated since their presence is required only by the compulsion to retain a parenthesized structure. The resulting set of 7 reductions is not only more compact, but it includes no reductions which are not standard axioms of group theory.

One of the most important deficiencies at the present time of complete sets of reductions is their inability to handle theories including commutativity. The reason for this is that complete sets of reductions are simplification based, and there is no way the rewriting rule x*y→y*x can be interpreted as a simplification for all instances of x and y.

The solution to this problem seems to lie in the area of, rather than having a single uniform complete set of reductions, having a hierarchical system composed of symmetric rewriting rules (such as for commutativity) and reductions. A term is irreducible if no reduction is applicable to any term derivable from the term by use of the symmetric rewriting rules. The unique termination criterion would be replaced by a check that the two irreducible terms be rewritable one to the other by the symmetric rewriting rules, i.e., that they be in the same equivalence class determined by the symmetric rewriting rules.

It seems essential that the output of application of the symmetric rewriting rules be equal in simplicity to the input. Otherwise there is a potential difficulty concerning termination of the simplification process.

In this area too, the use of associativity in a reduction causes complications. If x*y→y*x is taken to be a symmetric rewriting rule, then (a*a)*a and a*(a*a) are in the same equivalence class, but a*(a*a) is simpler than (a*a)*a in present complete sets of reductions.

Thus, extension of complete sets of reductions to handle commutativity may profit from using the unparenthesized representation for associative functions.

In addition, special unification algorithms (like the AC unification algorithm) may be usefully employed in future extensions to the Knuth-Bendix extension algorithm to handle commutativity or other properties. Such algorithms would be used rather than the ordinary unification algorithm in the generation of terms to which to reductions are both applicable.

3.20. Implementation

The semi-unification algorithms for vectors, bags, and classes, and the AC unification algorithm for associative-commutative functions of the first order predicate calculus have all been implemented substantially as described previously.

These semi-unification and unification algorithms replaced the standard predicate calculus unification algorithm of PSTP as described in the previous chapter. Thus, the new PSTP is capable of dealing with the data types vectors, bags, and classes, and

the corresponding concepts of associativity, commutativity, and idempotence at a high level.

The semi-unification procedures are modifiable (by the replacement of three functions) to use the QA4/QLISP variable designation conventions (i.e., variables to be assigned values must be preceded by one of the *, **, or *** prefixes) or the PSTP convention for the predicate calculus (i.e., variables are represented by (LISP) atoms while constants (as any function headed term) are represented by lists).

In the case where QA4/QLISP conventions are used, the type of unification to be performed is dictated by the type of the head of the expression. The type of expression heads (i.e., function symbols) can be set by the DEFTYPE function. For example, (DEFTYPE (QUOTE F) (QUOTE BAG)) declares F to have type bag so that the unification of (F *X *Y) and (F A B) will use bag semi-unification.

The default type is type vector, except BAG and CLASS are initially typed bag and class respectively.

In the case of predicate calculus representation, DEFTYPEing the function symbols is not sufficient. Since variables of the predicate calculus have no prefixes to distinguish what kind of matching can be performed with them (i.e., whether they are to be treated as simple or fragment variables), it is impossible to distinguisn textually the cases in which the expression f(xy) should be treated as if f were associative or not. (In the QA4/QLISP representation, associativity would be represented by

use of fragment variables rather than simple variables; in both cases the DEFTYPE of the function is the same.)

The solution in this implementation is the creation of three declarations: ASSOCIATIVE, COMMUTATIVE, and IDEMPOTENT.

The (ASSOCIATIVE f_1 ... f_n) function declares each of its argument function symbols f_i to be associative. The effect of this is to cause each variable occurrence as an argument of one of these functions to be treated as if the variable were preceded by the +++ (match at least one element) prefix. A function symbol not declared to be associative will have variable occurrences as arguments treated as if the variable were preceded by the + (match exactly one element) prefix.

The (COMMUTATIVE f_1 ... f_n) function declares each of its argument function symbols f_i to be commutative. The effect of this is essentially equivalent to declaring the type of the function symbol to be bag.

The (IDEMPOTENT f_1 ... f_n) function declares each of its argument function symbols f_i to be idempotent. The effect of this is essentially equivalent to declaring the type of the function symbol to be class.

If a function symbol is not declared in any of the above ways, it will be treated in the standard first order predicate calculus manner. If it is declared to be associative, all the variables will be implicitly +++ prefixed variables and unification will be associative. If it is declared to be commutative, but not associative, commutative unification will be performed, but a

variable can match only a single element of the opposing expression just as in the predicate calculus case. If it is declared to associative and commutative, both the previous effects will be present, representing associativity plus commutativity. Here it is optional whether bag semi-unification or AC unification is performed. If it is declared to be idempotent, class semi-unification will be performed (thus the idempotent declaration effectively does a commutative declaration as well since class unification is always commutative; there is no representation in the data types vector, bag, and class for idempotence without commutativity).

The ASSOCIATIVE, COMMUTATIVE, and IDEMPOTENT declarations can be undone by (NOT-ASSOCIATIVE f_1 ... f_n), (NOT-COMMUTATIVE f_1 ... f_n), and (NOT-IDEMPOTENT f_1 ... f_n) respectively.

3.21. Experimentation

Some preliminary experiments were performed to test the effectiveness of built-in associativity in problems such as the GROUP2 problem cited in the previous chapter and taken from the QA3.6 study.

The original axiomatization for the GROUP2 problem includes two 4 literal axioms which together represent associativity; each refers to one of the rearrangements of parentheses in the equality f(xf(yz))=f(f(xy)z.

This representation is clearly incompatible with our

representation for associativity which permits the declaration of particular function symbols to be associative. Accordingly, the GROUP2 problem can be reaxiomatized in the following way:

1.	e*x=z → x=z	add identity at left end of term
2.	x*e*y=z → x*y=z	add identity in middle of term
3.	x*e=z → x=z	add identity at right end of term
4.	x=z → e*x=z	remove identity at left end of term
5.	x*y=z → x*e*y=z	remove identity in middle of term
6.	x=z → x*e=z	remove identity at right end of term
7.	u*u*x=z → e*x=z	replace e by u*u at left end of term
8.	x*u*u*y=z → x*e*y=z	replace e by u*u in middle of term
9.	x*u*u=z → x*e=z	replace e by u*u at right end of term
10.	e*x=z → u*u*x=z	replace u*u by e at left end of term
11.	x*e*y=z → x*u*u*y=z	replace u*u by e in middle of term
12.	x*e=z → x*u*u=z	replace u*u by e at left end of term
13.	a*b=c	special hypothesis
14.	b*a=c	theorem

Note that although the equality symbol = is used in the above axiomatization, there is no need in this case for the special equality inference rules, and = was assigned no special interpretation in the experiment.

By simple inspection, it is apparent (since no antecedent literals are negated) that no negative literals can ever appear in a derived chain, and thus no reduction (requiring complementarity of literals, and thus requiring a negated literal) can occur. Hence, retention of A-literals is not necessary for completeness in this case. Therefore, retention of A-literals was suppressed to facilitate subsumption which would now have to check subsumption only of the single B-literal in each derived chain. Only a very weak form of subsumption was

employed, which checked for identity of the derived chains (after variable standardization so that variants (with literals in the same order) will be subsumed).

A very simple ordering function was employed which expanded chains in order of increasing size of their stored list representation. Extension by all axioms was employed at each step rather than just extension by a single axiom, i.e., SEARCH rather than SEARCH2 was used, a notably inefficient strategy. Nevertheless, the results were reasonably successful, with 11 chains being expanded (7 of which were on the solution path) resulting in the derivation of 147 chains 86 of which were eliminated by subsumption.

The resulting proof was:

```
14. b*a=c                 theorem
15. b*a*e=c               extend by 3
16. b*a*x1*x1=c           extend by 9
17. e*b*a*x1*x1=c         extend by 1
18. x2*x2*b*a*x1*x1=c     extend by 7
19. e*a*b=c               extend by 10
20. a*b=c                 extend by 4
21. Ø                     extend by 13
```

The most important step in the proof is the extension of chain 18 by axiom 10. Here, associative unification discovered the unifying substitution {x2←a, x1←a*b, u←a*a*b, x←a*b}.

This result can be substantially improved upon by (1) employing a more effective search strategy, including use of the SEARCH2 ratner than SEARCH command or (2) eliminating use of the identity element e, and axiomatizing the problem so that u*u can be directly inserted or removed anywnere in the term.

The deficiency of testing problems like GROUP2 witn built-in

associativity is the enormous change in representation. (An even larger and to a certain extent more appropriate change in representation would be use of equality inference rules permitting, for example, the replacement of axioms 1 through 6 by e*x=x and x*e=x.) This change of representation not only makes performance comparisons difficult, it will often bias the results against built-in associativity since the new representation produces only single literal derived chains which cannot be distinguished by length whereas the old representation produced length varying chains, use of the associative axioms adding 3 literals to a derived chain, and length preference strategies are among the most effective theorem proving search strategies.

In order to make a more meaningful assessment of the effectiveness of building in associativity and commutativity, we looked for a set of problems in which built-in associativity and commutativity could be turned on or off without altering the problem representation (except, of course, for the deletion of the associative or commutative axioms when associativity or commutativity were turned on).

A set of problems for manipulation of the arithmetic addition and subtraction functions for which this could be done was found in an experiment for the FORTRAN Deductive System (FDS) [59]. A predicate calculus axiomatization for these problems, originally expressed in terms of rewriting rules, was provided by Chang [9].

12 of the 18 FDS problems were run with two search strategies

and with (1) neither associativity nor commutativity built-in, (2) only commutativity built-in, (3) only associativity built-in, (4) both associativity and commutativity built-in. This is a total of 12*2*4=96 attempted proofs 83 of which were successful. With the exception of example 8, any attempted proof requiring the derivation of more than 500 chains was aborted. Not all strategies were tested on the remaining 6 of 16 FDS problems, but those that were resulted in no proof within the 500 chain limit. Thus, to conserve effort, only these 12 were fully examined.

Note that these results should only be used for examining the effectiveness of the building-in of associativity and commutativity. They are not comparable to the results of FDS since (1) the predicate calculus representation and rules of inferences differ from those of FDS which more closely resembles a paramodulation based system since rewriting rules can be applied to any subterm, (2) naive search strategies are being employed here by PSTP as opposed to the means-ends analysis of FDS, (3) FDS is a parameterized incomplete system whereas the search strategies employed by PSTP are complete (though it is not noted in [59] what effect the parameters had on the solution costs for FDS, and (4) all PSTP proofs used the same set of 11 axioms whereas FDS added as an axiom each previously proved theorem, (5) FDS employed learning to fill in values of a signature table to optimize its performance. (Since PSTP is itself a general symbol manipulation system, it could be used or adapted to possess these features of FDS. However, such use or adaptation was outside the scope of this experiment.)

	Acceptable Chains Generated							
	501/100 stragegy with built-in				nblits + size blits strategy with built-in			
Problem	None	Commu.	Assoc.	A.+C.	None	Commu.	Assoc.	A.+C.
1.	63	1	1	1	13	1	1	1
2.	3	3	3	3	3	3	3	3
3.	4	5	4	5	3	3	3	3
4.	280	1	82	1	155	1	23	1
5.	49	5	63	5	12	3	12	3
6.	>500	277	>500	260	>500	>500	>500	>500
7.	>500	21	17	17	95	45	74	8
8.	2615	5	>500	5	285	3	21	3
9.	48	1	59	1	12	1	12	1
10.	30	34	30	34	13	13	13	13
11.	>500	256	130	31	326	18	68	13
12.	>500	316	>500	292	>500	>500	490	44

Table 3.1

The results of this experiment are presented in Appendix B. A summary of the number of chains generated for each proof attempt is shown in Table 3.1. In the timing statistics, a time suffixed by $^{+}$ represents a time obtained using the version of PSTP described in the previous chapter which had a unification algorithm coded in assembly language and having no capability for built-in associativity or commutativity. All other times were obtained using a later version with a unification algorithm coded in LISP and allowing for associativity and commutativity. Comparison of the timing results show the latter version to be

approximately 2.5 times slower even when built-in associativity and commutativity are not being used.

The two search strategies used were the unbounded 501/100 strategy discussed in the previous chapter and a simple unbounded strategy ordering chains on the basis of expected size of stored list representation of B-literals. The latter strategy was generally though not uniformly superior for these examples. This is attributable to the importance of complexity of the generated terms (e.g., function depth) which is not considered by the 501/100 strategy which considers only length and deduction level.

The proofs discovered by the various combinations of search strategy and use of built-in associativity and commutativity are also presented in Appendix B.

Proofs requiring built-in associativity or commutativity are annotated by * or ** at the steps at which the proof would be blocked if built-in associativity or commutativity were turned off. Of course in the case of associativity, associativity is used in places other than just the noted lines since though the proof may not be blocked at a certain step, the step may require substitution for a variable using the associative convention of eliminating the extra variables, i.e., using f(abc) for f(f(ab)c). For example, chain 17 of proof C of example 4 has the term a+(b-c)+x1 in the last literal requiring built-in associativity be turned on lest the term be represented as (a+(b-c))+x1 or a+((b-c)+x1), but the proof would not be blocked by lack of built-in associativity until the following step.

Only rarely (example 3, 501/100 strategy, commutativity; example 5, 501/100 strategy, associativity; example 9, 501/100 strategy, associativity; example 10, 501/100 strategy, commutativity) did the use of built-in associativity or commutativity degrade the performance of PSTP on the problems, and never in a case where the commutative axiom was used in the nothing built-in case did building-in commuativity degrade performance, and likewise for associativity.

In examples 2, 3, and 10, no improvement was found using built-in associativity or commutativity, and the same proof was found for all combinations of search strategy and building-in associativity or commutativity. Neither the associative axiom nor the commutative axiom was used in any of these examples.

Far more typical are the results for the other problems in which substantial improvements were noted for either building-in either associativity or commutativity. For some examples and search strategies, the only proofs discovered within the 500 generated chains limit were discovered using built-in associativity or commutativity (examples 6, 7, 11, and 12). There was an up to 500 to 1 improvement in the number of chains generated (example 8) for using built-in associativity or commutativity.

Several proofs were substantially shortened by building-in associativity or commutativity, such as proofs of example 1, 4, and 9 being reduced to trivial one step proofs. In example 1, proof B represents the use of commutativity of + to prove one

associative rewrite derivable from the other! In some cases, use of built-in associativity improved performance even when the original proof contained no use of the associative axiom, as in example 8.

In summary, the use of built-in associativity or commutativity was highly effective for this set of problems, and the experiment strongly supports the claim of usefulness of building-in these concepts in this manner.

3.22. Background and summary

Material presented here concerning data types and representation of expressions is based on QLISP [37]. QLISP is an extension of LISP possessing most of the capability of QA4 [43], a language interpreted in LISP.

Specifically, the choice of fundamental data types of vectors, bags, and classes was made since these are the data types of QLISP. The syntax and most of the semantics for simple variables and for fragment variables with ++ and $$ prefixes follow QLISP rules. The association of a type with the head of each non-atomic expression is motivated by the QLISP DEFTYPEing capability; however, here the type is permanently associated with the head and affects future pattern matching or unification whereas DEFTYPEing in QLISP defined input/output conventions for expressions with some head.

The principal extension to data types and expression

representation presented here is the introduction of fragment variables with +++ and $$$ prefixes. Although the use of QLISP fragment variables permits the expression of variable length expressions and associativity, it is the use of the fragment variables with +++ and $$$ prefixes which permits their expression in a fashion identical to that of the formal logic. They also serve a useful role in eliminating some redundancy in unifying expressions after variable splitting.

Treatment of fragment variables is extended somewhat from that in QLISP. In QLISP, bags and classes may have only one fragment variable. Also, fragment variables in classes match only elements not matched elsewnere. This latter restriction is clearly incomplete and is thus eliminated here.

Incomplete unification algorithms similar to semi-unification have been developed for the first order predicate calculus case as for example by Nevins [34]. The separation of unification into two processes, semi-unification and variable splitting, to provide completeness was independently discovered by Slagle [46] in the first order predicate calculus case. In the case of argument lists of non-commutative associative functions, he defines a "weak associative unification algorithm" corresponding to our vector unification in the slightly more general vector case. An algorithm for commutativity is not explicitly specified, and the suggestion for combining associativity and commutativity is essentially to apply the weak associative unification algorithm to permuted argument lists of commutative

associative functions. This must be done with more care than is suggested by Slagle since an obvious commutative unification algorithm (permute the elements of just one of the expressions) followed by weak associative unification is not adequate since variables of the other expression would still be constrained to match contiguous sequences of elements of the opposing unpermuted expression.

Two forms of variable splitting were presented here. The first allowed variable splitting to be performed on only one of the expressions with the penalty that instances of that expression must be formed to which variable splitting is applied. These instances need only be formed in those cases where the variables being instantiated could be matched against expressions similar to the instances formed. The second requires variable splitting in both expressions and eliminates the formation of instances. The choice of variable splitting rules is dependent on the type of expressions which are most likely to be encountered. If fragment variables are always opposed by sequences of variables and fragment variable free expressions, then no instantiation is needed and variable splitting need only be applied to one of the expressions.

The variable splitting rule is called "widening" by Slagle and is applied to both expressions and is equivalent to the second form above. Variable splitting or widening corresponds to paramodulation by functional reflexive axioms in conventional mechanical theorem proving. The application of variable

splitting represents paramodulation by the functional reflexive axiom for tne function which is the head of the expression with the variable being split. The sometimes required instantiation of an expression by a vector, bag, or class subexpression which is then subjected to variable splitting (this may be necessary in tne case where variable splitting is applied to only one of the expressions, not to both) corresponds to paramodulation by the functional reflexive axiom for the function which is the head of the expression being added.

The complete unification algorithms for bags and classes presented here are new. These should be useful primarily in languages and applications for which completeness is very important and there is no advantage to having incomplete semi-unification around which merely returns the most likely to be useful unifiers.

In summary, it is our view that artificial intelligence languages should be designed in a manner which readily logical completeness. Tne use of incomplete pattern matching facilities in artificial intelligence languages is obviously a major cause of incompleteness in these languages. Accordingly, we have suggested a form for remedying the incompleteness aspect of pattern matching without excessively penalizing the language designer or user with the costs of ensuring completeness. This is accomplished by separating the pattern matching (unification) process into two components: the incomplete semi-unification process and the completeness providing variable splitting

process. We stated, and proved termination, soundness, and completeness (with variable splitting) for, semi-unification algorithms for the vector, bag, and class data types. In addition, we separately stated and proved complete unification algorithms for bags and classes.

4. Summary and Conclusion

This thesis takes a rather theoretically oriented look at what we call completeness issues in artificial intelligence languages. Two principal areas of incompleteness are noted: inference system and pattern matching. Each of these areas of incompleteness is treated in the most constructive fashion: we provide revised inference systems and pattern matching facilities which overcome the incompleteness. Further, this is done with a view to minimizing the added cost of running a logically complete versus incomplete system. Also, the results we provide are often of as much or even more significance to the field of theorem proving as they are to the design of artificial intelligence languages.

We feel we have contributed significantly to the solution of problems of logical incompleteness of artificial intelligence languages. The elimination of the incompleteness both facilitates use of these languages in their present applications and encourages their use in new domains, notably mathematical theorem proving. Our work in developing and testing special unification algorithms whose nature was suggested by artificial intelligence languages represents an especially significant and beneficial interaction of the fields of artificial intelligence languages and mechanical theorm proving. We believe the field of mechanical theorem proving can continue to make contributions to the design of artificial intelligence languages, and that a major contribution of artificial intelligence languages to the field of mechanical theorem proving (and many other domains) is the

continuing development of new control structures used in searching for proofs.

Some of the specific accomplishments of the thesis are summarized below.

4.1. An inference system for artificial intelligence languages

In the area of inference system completeness, we present the * procedure which is an abstraction of the MESON procedure which does not specify a conceptual execution form (AND/OR tree for MESON procedure, chains for linear MESON procedure). The * procedure is a direct extension of the problem reduction method which results in a complete theorem proving method for the first order predicate calculus.

In contrast to the MESON procedures in which expansion and contradiction (extension and reduction) are two different processes (where expansion is the same as the problem reduction operator and contradiction is the necessary augmentation of the problem reduction method), the * procedure is a uniform single inference operation procedure which continually applies the problem reduction operation in the context of the negations of all higher goals.

One way of interpreting the * procedure extension of the problem reduction method in terms of artificial intelligence languages is to say the * procedure results from the replacement of every (GOAL x) statment by a goal statement having the effect

of the statements (ASSERT ¬x) (GOAL x) or (DENY x) (GOAL x) where the assertion or denial is local only to the (GOAL x) attempt and does not cause consistency directed data base deletions. This certainly achieves logical completeness at minimal cost in organization, syntax, and efficiency of artificial intelligence languages.

The * and related procedures also have special equality inference rules deducible from the related rules of the model elimination with paramodulation procedure. Two distinct approaches to eliminating some of the problems of excessive generation of clauses in the equality inference rule system are discussed: a marking restriction which tries to prohibit alternate derivations of the same term by ordering the applications of equality inference rules and the extension of the concept of demodulation to the * procedure.

We also attempted to deal with the problem of the incompatibility of the use of models with the * procedure. So long as the problem reduction method was used, goals could be freely discarded without loss of as much completeness as exists for the problem reduction method if the goal was false in a model of the problem. This is no longer true when the more powerful * procedure is used. Paradoxically, it can even be shown that it is sometimes necessary to expand goals which are not only false in the model, but are false in the problem statement as well.

By relaxing restrictions on the * procedure, we provide a procedure which, although not yet proved complete, is capable of

handling all known counter-examples to the completeness of models in the * procedure.

4.2. Programmable Strategy Theorem Prover

The Programmable Strategy Theorem Prover (PSTP) was developed for its own sake and as a testbed for some of the ideas developed in the thesis.

A prime advantage of PSTP is the flexibility of specification of search and deletion strategy. Although the underlying inference system is fixed in contrast to some other theorem proving systems, the specification of search and deletion strategies is very general, relying on a general mechanism for attaching property values to chains (computed on need only) and using ordered numerical values of specified properties to sort chain lists and specify order of chain inference operations.

This simple provision for pure heuristic search [35] using ordering functions was itself a novelty in theorem provers at the time it was installed in PSTP. This facility was extended to allow partial expansion of chains using the SEARCH2 command.

Another novel characteristic of PSTP is its use of the linear MESON procedure rather than a form of resolution as its inference system. Although it is complete in contrast to natural deduction systems, the linear MESON, MESON, and * procedures (all variants of the same idea) retain the form of the problem reduction method. This problem reduction related form should make proofs

more comprehensible, especially when compared with proofs using non-linear resolution refinements. The linear MESON procedure also strongly supports the heuristic search mechanism of PSTP. Since derived chains do not interact with each other (inference operations are performed only on a chain or a chain and an axiom), the possible inference operations that can be performed on a chain are fixed (the chain can be extended by any axiom or be reduced). In non-linear resolution refinements, a derived clause could always be resolved against some future generated clause, never allowing for the possibility of its being removed from the search space because all possible inference operations had been performed on it.

We also analyzed the performance of PSTP to the performance of QA3.6 and an SL-resolution theorem prover. From this analysis, we deduced that (for that class of problems) PSTP with the linear MESON procedure performed competitively, that search strategies which give higher weight to length of clause relative to level of clause than admissable ones often outperform admissable strategies (at the risk of generating longer proofs), and that factorization as required by the SL-resolution procedure is probably a bad idea (we hold that the optionality of factorization in the model elimination procedure and its derivatives is a major advantage).

4.3. Unification algorithms for artificial intelligence languages

In the area of pattern matching, incompleteness arises in the failure to return all possible unifiers of two expressions. Depending on the nature of the expressions (such as the general pattern language of PLANNER), this problem may be an unsolvable one due to, for example, the undecidability of the emptiness of intersection of two context free languages or the solution of arbitrary groups with generators. Even simpler constructs (VECTORS of QA4/QLISP) have unification problems of unknown solubility (the word equation for monoids in this case of associativity only).

A general method of solution is to attempt to enumerate unifiers rather than produce them all at once. This is often more practical since there may be an infinite number of unifiers prohibiting their total generation and, since unifiers can often be generated in an order approaching simplest first order, and simple unifiers are more likely to be useful, effort can be saved by generating unifiers piecemeal.

The approach suggested here involves splitting the unification process up into two separate processes: semi-unification and elaboration. The semi-unification process returns for a pair of expressions (one possibly restricted in form) a finite number of unifiers. This set of unifiers need not be a complete set. Semi-unification should return the most obvious unifiers of the

two expressions. This finite unification component is augmented by the potentially infinite elaboration process which alters one (the possibly restricted form) expression causing the semi-unification algorithm to produce new unifiers using the new expression. Alternating applications of elaboration and semi-unification would enumerate all unifiers.

Three special cases of great interest are discussed in substantial detail: the QA4/QLISP VECTOR (associativity), BAG (associativity and commutativity), and CLASS (associativity, commutativity, and idempotence (sets)) data types. Here, semi-unification is a unification process restricted to not splitting up the values of variables. The elaboration procedure (actually a slight variant of the idea of elaboration) consists of splitting fragment variables into two fragment variables.

In addition, we provide complete unification algorithms for the BAG and CLASS cases. The complete unification for BAGs is also extended to the slightly more complex case of unification in the first order predicate calculus with associativity and commutativity (the AC unification algorithm). We regard the AC unification algorithm and the related BAG unification algorithm as having substantial significance.

Experimentation with a version of PSTP augmented by the special unification procedures for associativity and commutativity confirms the view that substantial gains in performance can be achieved by such means.

In parallel with this work on unification algorithms there has

been significant work done by Livesey and Siekmann in the same area. They have been concerned with the problem of associative (vector) unification [62] and have also developed algorithms for commutativity (equivalent to bag semi-unification with no fragment variables), idempotence, and associativity plus commutativity plus idempotence (classes) [63]. Commutativity and idempotence occur infrequently without associativity in the problems with which we are concerned, so Livesey and Siekmann's most important work to us is that concerned with decidable cases of associative (vector) unification and the associativity plus commutativity plus idempotence (class) case. While associative-commutative or bag unification depends on the solution of a homogeneous linear Diophantine equation, they make the perceptive observation that class unification operates in the same fashion except the solution of the equation uses Boolean algebra addition rather than ordinary arithmetic addition. This results in a cleaner, and probably more efficient, class unification algorithm than the one presented here.

Significant developments have also occurred in the area of complete sets of reductions discussed briefly in section 3.19. Lankford and Ballantyne have developed complete sets of reductions for commutativity without associativity and a semi-decision procedure for incompleteness of sets of reductions using permutation equations [64,65]. Both are based on the concept of a hierarchical approach to complete sets of reductions including reductions and an equivalence class specified by

commutative or permutative axioms. Huet [61] also developed this approach to a decision (not just semi-decision) procedure for reductions with an equivalence class where the reductions are restricted to not having duplicated variables in their left hand sides (left free). Stickel and Peterson have developed and implemented the idea of using complete unification algorithms directly in computing complete sets of reductions rather than using an equivalence class [49]. Lankford and Ballantyne also developed some of the same techniques for application to associative-commutative reductions and used associative-commutative unification for testing completeness of the set of reductions though not for generating new reductions [66].

Appendix A. PSTP Performance Study Results

A.1. Examples

1. BURSTALL Example

Axioms:

1. has(p1,ass(j,n0))
2. follows(p2,p1)
3. has(p2,ass(k,n1))
4. labels(loop,p3)
5. follows(p3,p2)
6. has(p3,ifthen(equal(j,n),p4))
7. has(p4,goto(out))
8. follows(p5,p4)
9. follows(p6,p3)
10. has(p6,ass(k,times(n2,k)))
11. follows(p7,p6)
12. has(p7,ass(j,plus(j,n1)))
13. follows(p8,p7)
14. has(p8,goto(loop))
15. follows(xp,yp) → succeeds(xp,yp)
16. succeeds(xp,zp) succeeds(zp,yp) → succeeds(xp,yp)
17. has(xp,goto(zp)) labels(zp,yp) → succeeds(yp,xp)
18. has(xp,ifthen(ze,yp)) → succeeds(yp,xp)

Theorem:

19. succeeds(p3,p3)

2. SHORTBURST Example

Axioms:

1. labels(loop,p3)
2. has(p3,ifthen(equal(j,n),p4))
3. has(p4,goto(out))
4. follows(p5,p4)
5. follows(p8,p3)
6. has(p8,goto(loop))
7. follows(xp,yp) → succeeds(xp,yp)
8. succeeds(xp,zp) succeeds(zp,yp) → succeeds(xp,yp)
9. has(xp,goto(zp)) labels(zp,yp) → succeeds(yp,xp)
10. has(xp,ifthen(ze,yp)) → succeeds(yp,xp)

Theorem:

11. succeeds(p3,p3)

3. HAS-PARTS Example 1

Axioms:

1. in(John,boy)
2. in(x,boy) → in(x,human)
3. hp(x,xm,y) → in(skl(x,y,z,xm,xn),y) hp(x,t(xm,xn),z)
4. hp(x,xm,y) → ¬hp(skl(x,y,z,xm,xn),xn,z) hp(x,t(xm,xn),z)
5. in(x,hand) → hp(x,n5,fingers)
6. in(x,human) → hp(x,n2,arm)
7. in(x,arm) → hp(x,n1,hand)

Theorem:

8. hp(John,t(n2,n1),hand)

4. HAS-PARTS Example 2

Theorem:

8. hp(John,t(t(n2,n1),n5),fingers)

5. PRIM Example

Axioms:

1. Dxx
2. Dxy Dyz → Dxz
3. Px Dg(x)x
4. Px Lnlg(x)
5. Px Lg(x)x
6. Lnlx Lxa → Pf(x)
7. Lnlx Lxa → Df(x)x
8. Lnla

Theorem:

(9. ¬Px ¬Dxa negation of theorem)
10. Px1 Dx1a

6. ANCES1 Example

Axioms:

1. ¬J A H
2. K H J
3. ¬K H J
4. ¬A ¬B
5. ¬A B
6. ¬H ¬C

Theorem:

7. H ¬C

7. NUM1 Example

Axioms:

1. Px Myzw Dxw → Dxy Dxz
2. Mxxs(x)
3. Mxyz → Myxz
4. Mxyz → Dxz
5. Mas(c)s(b)
6. Pa

Theorem:

7. Dab

8. GROUP1 Example

Axioms:

1. Pxyu Pyzv Pxvw → Puzw
2. Pxyu Pyzv Puzw → Pxvw
3. Pg(xy)xy
4. Pxh(xy)y
5. Pxyf(xy)

Theorem:

(6. ¬Pj(x)xj(x) negation of theorem)
7. Pj(xl)xlj(xl)

9. GROUP2 Example

Axioms:

1. Pxex
2. Pexx
3. Pxyu Pyzv Puzw → Pxvw
4. Pxyu Pyzv Pxvw → Puzw
5. Pxxe
6. Pabc

Theorem:

7. Pbac

A.2. Statistics

Length Multiplier	Level Multiplier	Length Maximum	Level Maximum	Proof Code	Chains Generated (ret/acc/tot/att)	Search Time (sec)
1. BURSTALL Example						
101	100	-	-	A	191/191/215/3129	46.3
501	100	-	-	B	74/ 74/ 75/1229	16.1
101	100	2	12	A	45/ 45/ 45/741	9.8
501	100	2	12	A	45/ 45/ 45/741	9.2
QA3.6 best		2	12		38/ 42/ /1462	
QA3.6 mean 16/19		2	12		99/118/ /3222	
SLRTP		3	13		48/ 48/ /122	
2. SHORTBURST Example						
101	100	-	-	A	18/ 18/ 19/144	3.0
501	100	-	-	A	16/ 16/ 16/128	2.1
101	100	2	10	A	16/ 16/ 16/128	2.2
501	100	2	10	A	16/ 16/ 16/128	2.1
QA3.6 best		2	10		12/ 12/ /255	
QA3.6 mean 14/14		2	10		20/ 21/ /325	
SLRTP		3	10		16/ 16/ / 42	
3. HAS-PARTS Example 1						
Implicative form for axioms:						
101	100	-	-	A	7/ 7/ 7/ 47	0.7
501	100	-	-	A	7/ 7/ 7/ 47	0.7
101	100	3	10	A	7/ 7/ 7/ 47	1.0
501	100	3	10	A	7/ 7/ 7/ 47	0.8
Disjunctive form for axioms:						
101	100	-	-	A	12/ 12/ 12/124	1.8
501	100	-	-	A	12/ 12/ 12/124	1.7
101	100	3	10	A	12/ 12/ 12/124	1.5
501	100	3	10	A	12/ 12/ 12/124	1.5
QA3.6 best		2	10		8/ 10/ /112	
QA3.6 mean 6/6		2	10		20/ 24/ /343	
SLRTP		2	10		12/ 12/ / 29	

Length Multiplier	Level Multiplier	Length Maximum	Level Maximum	Proof Code	Chains Generated (ret/acc/tot/att)	Search Time (sec)
4. HAS-PARTS Example 2						
Implicative form for axioms:						
101	100	-	-	A	11/ 11/ 11/ 83	1.3
501	100	-	-	A	11/ 11/ 11/ 83	1.3
101	100	3	10	A	11/ 11/ 11/ 83	1.3
501	100	3	10	A	11/ 11/ 11/ 83	1.9
Disjunctive form for axioms:						
101	100	-	-	A	50/ 50/ 50/478	8.1
501	100	-	-	B	38/ 38/ 38/430	5.7
101	100	3	10	A	38/ 38/ 38/430	5.7
501	100	3	10	B	38/ 38/ 38/430	6.4
QA3.6 best		2	10		12/ 14/ /205	
QA3.6 mean 6/7		2	10		44/ 51/ /938	
SLRTP		3	13		20/ 20/ / 41	
5. PRIM Example						
Implicative form for axioms:						
101	100	-	-	A	812/812/1052/8072	200.7
501	100	-	-	A	57/ 57/ 64/640	11.6
101	100	3	18	A	70/ 70/ 82/816	12.3
501	100	3	18	A	54/ 54/ 60/609	8.2
Disjunctive form for axioms:						
101	100	-	-	B	165/165/187/1883	28.5
501	100	-	-	C	101/101/113/1220	18.8
101	100	3	10	B	130/130/146/1532	20.2
501	100	3	10	B	130/130/146/1532	18.8
101	100	3	18	B	130/130/146/1532	21.7
501	100	3	18	C	101/101/113/1220	16.8
QA3.6 best		3	10		13/ 19/ /208	
QA3.6 mean 9/10		3	10		36/ 97/ /999	
SLRTP		3	11		122/134/ /243	
6. ANCES1 Example						
101	100	-	-	A	23/ 23/ 23/240	3.2
501	100	-	-	A	13/ 13/ 13/108	1.3
101	100	2	10	A	13/ 13/ 13/108	1.6
501	100	2	10	A	13/ 13/ 13/108	1.4
QA3.6 best		2	10		5/ 12/ /158	
QA3.6 mean 19/20		2	10		6/ 13/ /129	
SLRTP		3	10		14/ 14/ / 26	

Length Multiplier	Level Multiplier	Length Maximum	Level Maximum	Proof Code	Chains Generated (ret/acc/tot/att)	Search Time (sec)
7. NUM1 Example						
101	100	-	-	A	10/ 10/ 11/ 47	1.2
501	100	-	-	A	10/ 10/ 11/ 47	1.1
101	100	5	10	A	10/ 10/ 11/ 47	0.9
501	100	5	10	A	10/ 10/ 11/ 47	0.9
QA3.6 best		5	10		8/ 10/ / 68	
QA3.6 mean 11/11		5	10		9/ 11/ / 83	
SLRTP		5	10		9/ 9/ / 21	
8. GROUP1 Example						
101	100	-	-	A	14/ 14/ 14/ 54	2.0
501	100	-	-	A	14/ 14/ 14/ 54	1.5
101	100	3	10	A	14/ 14/ 14/ 54	1.5
501	100	3	10	A	14/ 14/ 14/ 54	1.4
QA3.6 best		3	10		7/ 7/ / 33	
QA3.6 mean 9/9		3	10		7/ 7/ / 34	
SLRTP		3	10		9/ 12/ / 35	
9. GROUP2 Example						
101	100	-	-	A	576/576/752/2408	97.2
501	100	-	-	B	119/119/149/500	17.0
101	100	4	10	A	225/225/325/938	28.8
501	100	4	10	A	225/225/325/938	36.7
QA3.6 best		3	10		54/ 74/ /324	
QA3.6 mean 8/8		3	10		60/ 82/ /517	
SLRTP		?	?		no proof found	

A.3. Proofs

1. BURSTALL Example

Proof A

19.	succeeds(p3,p3)	theorem
20.	[succeeds(p3,p3)] succeeds(p3,x1) succeeds(x1,p3)	extend by 16
21.	[succeeds(p3,p3)] succeeds(p3,x1) [succeeds(x1,p3)] follows(x1,p3)	extend by 15
22.	[succeeds(p3,p3)] succeeds(p3,p6)	extend by 9
23.	[succeeds(p3,p3)] [succeeds(p3,p6)] succeeds(p3,x1) succeeds(x1,p6)	extend by 16
24.	[succeeds(p3,p3)] [succeeds(p3,p6)] succeeds(p3,x1) [succeeds(x1,p6)] follows(x1,p6)	extend by 15
25.	[succeeds(p3,p3)] [succeeds(p3,p6)] succeeds(p3,p7)	extend by 11
26.	[succeeds(p3,p3)] [succeeds(p3,p6)] [succeeds(p3,p7)] succeeds(p3,x1) succeeds(x1,p7)	extend by 16
27.	[succeeds(p3,p3)] [succeeds(p3,p6)] [succeeds(p3,p7)] succeeds(p3,x1) [succeeds(x1,p7)] follows(x1,p7)	extend by 15
28.	[succeeds(p3,p3)] [succeeds(p3,p6)] [succeeds(p3,p7)] succeeds(p3,p8)	extend by 13
29.	[succeeds(p3,p3)] [succeeds(p3,p6)] [succeeds(p3,p7)] [succeeds(p3,p8)] has(p8,goto(x1)) labels(x1,p3)	extend by 17
30.	[succeeds(p3,p3)] [succeeds(p3,p6)] [succeeds(p3,p7)] [succeeds(p3,p8)] has(p8,goto(loop))	extend by 4
31.	∅	extend by 14

Proof B

19.	succeeds(p3,p3)	theorem
20.	[succeeds(p3,p3)] succeeds(p3,x1) succeeds(x1,p3)	extend by 16
21.	[succeeds(p3,p3)] succeeds(p3,x1) [succeeds(x1,p3)] succeeds(x1,x2) succeeds(x2,p3)	extend by 16
22.	[succeeds(p3,p3)] succeeds(p3,x1) [succeeds(x1,p3)] succeeds(x1,x2) [succeeds(x2,p3)] follows(x2,p3)	extend by 15
23.	[succeeds(p3,p3)] succeeds(p3,x1) [succeeds(x1,p3)] succeeds(x1,p6)	extend by 9
24.	[succeeds(p3,p3)] succeeds(p3,x1) [succeeds(x1,p3)] [succeeds(x1,p6)] follows(x1,p6)	extend by 15
25.	[succeeds(p3,p3)] succeeds(p3,p7)	extend by 11
26.	[succeeds(p3,p3)] [succeeds(p3,p7)] succeeds(p3,x1) succeeds(x1,p7)	extend by 16
27.	[succeeds(p3,p3)] [succeeds(p3,p7)] succeeds(p3,x1) [succeeds(x1,p7)] follows(x1,p7)	extend by 15
28.	[succeeds(p3,p3)] [succeeds(p3,p7)] succeeds(p3,p8)	extend by 13
29.	[succeeds(p3,p3)] [succeeds(p3,p7)] [succeeds(p3,p8)] has(p8,goto(x1)) labels(x1,p3)	extend by 17
30.	[succeeds(p3,p3)] [succeeds(p3,p7)] [succeeds(p3,p8)] has(p8,goto(loop))	extend by 4
31.	∅	extend by 14

2. SHORTBURST Example

Proof A

11.	succeeds(p3,p3)	theorem
12.	[succeeds(p3,p3)] succeeds(p3,x1) succeeds(x1,p3)	extend by 8
13.	[succeeds(p3,p3)] succeeds(p3,x1) [succeeds(x1,p3)] follows(x1,p3)	extend by 7
14.	[succeeds(p3,p3)] succeeds(p3,p8)	extend by 5
15.	[succeeds(p3,p3)] [succeeds(p3,p8)] has(p8,goto(x1)) labels(x1,p3)	extend by 9
16.	[succeeds(p3,p3)] [succeeds(p3,p8)] has(p8,goto(loop))	extend by 1
17.	∅	extend by 6

3. HAS-PARTS Example 1

Proof A

8.	hp(John,t(n2,n1),hand)	theorem

```
 9. [hp(John,t(n2,n1),hand)] hp(John,n2,x1) hp(skl(John,x1,hand,n2,n1),n1,hand)                    extend by 4a
10. [hp(John,t(n2,n1),hand)] hp(John,n2,x1) [hp(skl(John,x1,hand,n2,n1),n1,hand)]
    in(skl(John,x1,hand,n2,n1),arm)                                                               extend by 7
11. [hp(John,t(n2,n1),hand)] hp(John,n2,arm)                                                      extend by 3b
12. [hp(John,t(n2,n1),hand)] [hp(John,n2,arm)] in(John,human)                                     extend by 6
13. [hp(John,t(n2,n1),hand)] [hp(John,n2,arm)] [in(John,human)] in(John,boy)                      extend by 2
14. ∅                                                                                             extend by 1
```

4. HAS-PARTS Example 2

Proof A

```
 8. hp(John,t(t(n2,n1),n5),fingers)                                                               theorem
 9. [hp(John,t(t(n2,n1),n5),fingers)] hp(John,t(n2,n1),x1)
    hp(skl(John,x1,fingers,t(n2,n1),n5),n5,fingers)                                               extend by 4a
10. [hp(John,t(t(n2,n1),n5),fingers)] hp(John,t(n2,n1),x1)
    [hp(skl(John,x1,fingers,t(n2,n1),n5),n5,fingers)] in(skl(John,x1,fingers,t(n2,n1),n5),hand)    extend by 5
11. [hp(John,t(t(n2,n1),n5),fingers)] hp(John,t(n2,n1),hand)                                      extend by 3b
12. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] hp(John,n2,x1)
    hp(skl(John,x1,hand,n2,n1),n1,hand)                                                           extend by 4a
13. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] hp(John,n2,x1)
    [hp(skl(John,x1,hand,n2,n1),n1,hand)] in(skl(John,x1,hand,n2,n1),arm)                          extend by 7
14. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] hp(John,n2,arm)                    extend by 3b
15. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] [hp(John,n2,arm)] in(John,human)   extend by 6
16. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] [hp(John,n2,arm)] [in(John,human)]
    in(John,boy)                                                                                  extend by 2
17. ∅                                                                                             extend by 1
```

Proof B

```
 8. hp(John,t(t(n2,n1),n5),fingers)                                                               theorem
 9. [hp(John,t(t(n2,n1),n5),fingers)] hp(John,t(n2,n1),x1) ¬in(skl(John,x1,fingers,t(n2,n1),n5),x1)  extend by 3a
10. [hp(John,t(t(n2,n1),n5),fingers)] hp(John,t(n2,n1),hand)
    [¬in(skl(John,hand,fingers,t(n2,n1),n5),hand)] ¬hp(skl(John,hand,fingers,t(n2,n1),n5),n5,fingers)  extend by 5b
11. [hp(John,t(t(n2,n1),n5),fingers)] hp(John,t(n2,n1),hand)                                      extend by 4b
12. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] hp(John,n2,x1)
    hp(skl(John,x1,hand,n2,n1),n1,hand)                                                           extend by 4a
13. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] hp(John,n2,x1)
    [hp(skl(John,x1,hand,n2,n1),n1,hand)] in(skl(John,x1,hand,n2,n1),arm)                          extend by 7
14. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] hp(John,n2,arm)                    extend by 3b
15. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] [hp(John,n2,arm)] in(John,human)   extend by 6
16. [hp(John,t(t(n2,n1),n5),fingers)] [hp(John,t(n2,n1),hand)] [hp(John,n2,arm)] [in(John,human)]
    in(John,boy)                                                                                  extend by 2
17. ∅                                                                                             extend by 1
```

5. PRIM Example

Proof A

```
10. Px1 Dx1a                                                                                     theorem
11. Px1 [Dx1a] Dx1x2 Dx2a                                                                        extend by 2
12. Px1 [Dx1a] Dx1g(a) [Dg(a)a] ¬Pa                                                              extend by 3a
13. Px1 [Dx1a] Dx1g(a) [Dg(a)a] [¬Pa] Daa                                                        extend by 9b
14. Px1 [Dx1a] Dx1g(a)                                                                           extend by 1
15. Pf(g(a)) [Df(g(a))a] [Df(g(a))g(a)] Lnlg(a) Lg(a)a                                           extend by 7
16. Pf(g(a)) [Df(g(a))a] [Df(g(a))g(a)] Lnlg(a) [Lg(a)a] ¬Pa                                     extend by 5a
17. Pf(g(a)) [Df(g(a))a] [Df(g(a))g(a)] Lnlg(a) [Lg(a)a] [¬Pa] Daa                               extend by 9b
```

```
    18. Pf(g(a)) [Df(g(a))a] [Df(g(a))g(a)] Lnlg(a)                                                    extend by 1
    19. Pf(g(a)) [Df(g(a))a] [Df(g(a))g(a)] [Lnlg(a)] ¬Pa                                              extend by 4a
    20. Pf(g(a)) [Df(g(a))a] [Df(g(a))g(a)] [Lnlg(a)] [¬Pa] Daa                                        extend by 9b
    21. Pf(g(a))                                                                                       extend by 1
    22. [Pf(g(a))] Lnlg(a) Lg(a)a                                                                      extend by 6
    23. [Pf(g(a))] Lnlg(a) [Lg(a)a] ¬Pa                                                                extend by 5a
    24. [Pf(g(a))] Lnlg(a) [Lg(a)a] [¬Pa] Daa                                                          extend by 9b
    25. [Pf(g(a))] Lnlg(a)                                                                             extend by 1
    26. [Pf(g(a))] [Lnlg(a)] ¬Pa                                                                       extend by 4a
    27. [Pf(g(a))] [Lnlg(a)] [¬Pa] Daa                                                                 extend by 9b
    28. ∅                                                                                              extend by 1

Proof B
    10. Px1 Dx1a                                                                                       theorem
    11. Pa                                                                                             extend by 1
    12. [Pa] ¬Lnlg(a)                                                                                  extend by 4b
    13. [Pa] [¬Lnlg(a)] Lg(a)a ¬Pf(g(a))                                                               extend by 6c
    14. [Pa] [¬Lnlg(a)] Lg(a)a [¬Pf(g(a))] Df(g(a))a                                                   extend by 9b
    15. [Pa] [¬Lnlg(a)] Lg(a)a [¬Pf(g(a))] [Df(g(a))a] Df(g(a))x1 Dx1a                                 extend by 2
    16. [Pa] [¬Lnlg(a)] Lg(a)a [¬Pf(g(a))] [Df(g(a))a] Df(g(a))g(a)                                    extend by 3a
    17. [Pa] [¬Lnlg(a)] Lg(a)a                                                                         extend by 7
    18. ∅                                                                                              extend by 5a

Proof C
    10. Px1 Dx1a                                                                                       theorem
    11. Pa                                                                                             extend by 1
    12. [Pa] -Dg(a)a                                                                                   extend by 3b
    13. [Pa] [-Dg(a)a] Dx1g(a) -Dx1a                                                                   extend by 2b
    14. [Pa] [-Dg(a)a] Dx1g(a) [-Dx1a] Px1                                                             extend by 9a
    15. [Pa] [-Dg(a)a] Df(x1)g(a) [-Df(x1)a] [Pf(x1)] Lnlx1 Lx1a                                       extend by 6
    16. [Pa] [-Dg(a)a] Df(g(a))g(a) [-Df(g(a))a] [Pf(g(a))] Lnlg(a)                                    extend by 5a
    17. [Pa] [-Dg(a)a] Df(g(a))g(a)                                                                    extend by 4a
    18. [Pa] [-Dg(a)a] [Df(g(a))g(a)] Lnlg(a) Lg(a)a                                                   extend by 7
    19. [Pa] [-Dg(a)a] [Df(g(a))g(a)] Lnlg(a)                                                          extend by 5a
    20. ∅                                                                                              extend by 4a
```

6. ANCES1 Example

```
Proof A
     7. H ¬C                                                                                           theorem
     8. H                                                                                              extend by 6a
     9. [H] J ¬A                                                                                       extend by 1a
    10. [H] J [¬A] B                                                                                   extend by 4b
    11. [H] J                                                                                          extend by 5a
    12. [H] [J] ¬K                                                                                     extend by 2a
    13. ∅                                                                                              extend by 3c
```

7. NUM1 Example

```
Proof A
     7. Dab                                                                                            theorem
     8. [Dab] Pa Mx1bx2 Dax2 ¬Dax1                                                                     extend by 1a
     9. [Dab] Pa Mbbx1 Dax1                                                                            reduce
    10. [Dab] Pa Mbbx1 [Dax1] Max2x1                                                                   extend by 4
```

```
    11. [Dab] Pa Mbbs(b)                                                    extend by 5
    12. [Dab] Pa                                                            extend by 2
    13. ∅                                                                   extend by 6

8. GROUP1 Example

   Proof A
     7. Pj(x1)x1j(x1)                                                       theorem
     8. [Pj(x1)x1j(x1)] Px2x3j(x1) Px3x1x4 Px2x4j(x1)                       extend by 1
     9. [Pj(x1)x1j(x1)] Pg(x2j(x1))x3j(x1) Px3x1x2                          extend by 3
    10. [Pj(h(x1x2))h(x1x2)j(h(x1x2))] Pg(x2j(h(x1x2)))x1j(h(x1x2))         extend by 4
    11. ∅                                                                   extend by 3

9. GROUP2 Example

   Proof A
     7. Pbac                                                                theorem
     8. [Pbac] Pbx1x2 Px1x3a Px2x3c                                         extend by 3
     9. [Pbac] Pbx1e Px1ca                                                  extend by 2
    10. [Pbac] Pbx1e [Px1ca] Px2x3x1 Px3cx4 Px2x4a                          extend by 4
    11. [Pbac] Pbx1e [Px1ca] Pax2x1 Px2ce                                  extend by 1
    12. [Pbac] Pbx1e [Px1ca] Pacx1                                         extend by 5
    13. [Pbac] Pbx1e [Px1ca] [Pacx1] Pax2x3 Px2x4c Px3x4x1                 extend by 3
    14. [Pbac] Pbx1e [Px1ca] [Pacx1] Pax2e Px2x1c                          extend by 2
    15. [Pbac] Pbbe [Pbca] [Pacb] Paae                                     extend by 6
    16. [Pbac] Pbbe                                                        extend by 5
    17. ∅                                                                  extend by 5

   Proof B
     7. Pbac                                                                theorem
     8. [Pbac] Pbx1x2 Px1x3a Px2x3c                                         extend by 3
     9. [Pbac] Pbx1a Px1ba                                                  extend by 6
    10. [Pbac] Pbx1a [Px1ba] Px2x3x1 Px3bx4 Px2x4a                          extend by 4
    11. [Pbac] Pbx1a [Px1ba] Pax2x1 Px2be                                  extend by 1
    12. [Pbac] Pbx1a [Px1ba] Pabx1                                         extend by 5
    13. [Pbac] Pbca                                                        extend by 6
    14. [Pbac] [Pbca] Px1x2b Px2cx3 Px1x3a                                 extend by 4
    15. [Pbac] [Pbca] Pax1b Px1ce                                          extend by 1
    16. [Pbac] [Pbca] Pacb                                                 extend by 5
    17. [Pbac] [Pbca] [Pacb] Pax1x2 Px1x3c Px2x3b                          extend by 3
    18. [Pbac] [Pbca] [Pacb] Pax1e Px1bc                                   extend by 2
    19. [Pbac] [Pbca] [Pacb] Paae                                          extend by 6
    20. ∅                                                                  extend by 5
```

Appendix B. Associativity and Commutativity Test Results

B.1. Examples

1. Quinlan & Hunt Example 1 (Chang Example 10a)

Axioms:

```
 1. x+y→y+x
 2. x+(y+z)→(x+y)+z
 3. (x+y)-y→x
 4. x→(x+y)-y
 5. (x-y)+z→(x+z)-y
 6. (x+y)-z→(x-z)+y
 7. x→y  y→z → x→z
 8. x→x
 9. x→y  u→x+v → u→y+v
10. x→y  u→x-v → u→y-v
11. x→y  u→v-x → u→v-y
```

Theorem:

```
12. (a+b)+c→a+(b+c)
```

2. Quinlan & Hunt Example 2

Theorem:

```
12. (a-b)+b→a
```

3. Quinlan & Hunt Example 3

Theorem:

```
12. a→(a-b)+b
```

4. Quinlan & Hunt Example 4

Theorem:

```
12. a+(b-c)→(a+b)-c
```

5. Quinlan & Hunt Example 5 (Chang Example 10b)

Theorem:

```
12. (a-b)+c→a+(c-b)
```

6. Quinlan & Hunt Example 6

Theorem:

```
12. (a-c)-(b-c)→a-b
```

7. Quinlan & Hunt Example 7

Theorem:

```
12. (a+c)-(b+c)→a-b
```

8. Quinlan & Hunt Example 8 (Chang Example 10c)

Theorem:
12. a+(b-c)→(a-c)+b

9. Quinlan & Hunt Example 9 (Chang Example 10d)

Theorem:
12. (a+b)-c→a+(b-c)

10. Quinlan & Hunt Example 10

Theorem:
12. (a-b)-c→(a-c)-b

11. Quinlan & Hunt Example 11

Theorem:
12. a-(b+c)→(a-b)-c

12. Quinlan & Hunt Example 18

Theorem:
12. (a+b)-c→a-(c-b)

B.2. Statistics

Search Strategy Employed/ Options Used	Proof Code	Chains Generated (ret/acc/tot/att)	Search Time (sec)
1. Quinlan & Hunt Example 1 (Chang Example 10a)			
501*nblit + 100*level	A	63/ 63/ 86/488	12.8+
" - axiom 1 + commutativity	B	1/ 1/ 1/ 6	0.7
" - axiom 2 + associativity	C	1/ 1/ 1/ 1	0.2
" - axioms 1&2 + commu. + assoc.	C	1/ 1/ 1/ 1	0.2
nblit + size blits	A	13/ 13/ 18/ 87	4.7
" - axiom 1 + commutativity	B	1/ 1/ 1/ 6	0.6
" - axiom 2 + associativity	C	1/ 1/ 1/ 1	0.2
" - axioms 1&2 + commu. + assoc.	C	1/ 1/ 1/ 1	0.2
2. Quinlan & Hunt Example 2			
501*nblit + 100*level	A	3/ 3/ 4/ 19	1.3
" - axiom 1 + commutativity	A	3/ 3/ 4/ 18	1.1
" - axiom 2 + associativity	A	3/ 3/ 4/ 18	0.8
" - axioms 1&2 + commu. + assoc.	A	3/ 3/ 4/ 17	1.3
nblit + size blits	A	3/ 3/ 4/ 18	1.1
" - axiom 1 + commutativity	A	3/ 3/ 4/ 16	1.2
" - axiom 2 + associativity	A	3/ 3/ 4/ 16	0.8
" - axioms 1&2 + commu. + assoc.	A	3/ 3/ 4/ 14	1.0
3. Quinlan & Hunt Example 3			
501*nblit + 100*level	A	4/ 4/ 5/ 24	1.0
" - axiom 1 + commutativity	A	5/ 5/ 6/ 28	1.4
" - axiom 2 + associativity	A	4/ 4/ 5/ 22	1.2
" - axioms 1&2 + commu. + assoc.	A	5/ 5/ 6/ 25	1.5
nblit + size blits	A	3/ 3/ 4/ 14	0.7
" - axiom 1 + commutativity	A	3/ 3/ 4/ 13	0.7
" - axiom 2 + associativity	A	3/ 3/ 4/ 13	1.0
" - axioms 1&2 + commu. + assoc.	A	3/ 3/ 4/ 12	0.8
4. Quinlan & Hunt Example 4			
501*nblit + 100*level	A	280/280/378/2154	140.2
" - axiom 1 + commutativity	B	1/ 1/ 1/ 3	0.3
" - axiom 2 + associativity	C	82/ 82/110/547	54.0
" - axioms 1&2 + commu. + assoc.	B	1/ 1/ 1/ 3	0.3
nblit + size blits	A	155/155/224/1154	83.8
" - axiom 1 + commutativity	B	1/ 1/ 1/ 3	0.3
" - axiom 2 + associativity	D	23/ 23/ 32/152	11.9
" - axioms 1&2 + commu. + assoc.	B	1/ 1/ 1/ 3	0.3

Search Strategy Employed/ Options Used	Proof Code	Chains Generated (ret/acc/tot/att)	Search Time (sec)
5. Quinlan & Hunt Example 5 (Chang Example 10b)			
501*nblit + 100*level	A	49/ 49/ 64/373	9.3+
" - axiom 1 + commutativity	B	5/ 5/ 6/ 27	2.2
" - axiom 2 + associativity	A	63/ 63/ 80/413	33.9
" - axioms 1&2 + commu. + assoc.	B	5/ 5/ 6/ 24	2.0
nblit + size blits	A	12/ 12/ 17/ 85	5.7
" - axiom 1 + commutativity	B	3/ 3/ 4/ 12	0.9
" - axiom 2 + associativity	A	12/ 12/ 17/ 74	5.5
" - axioms 1&2 + commu. + assoc.	B	3/ 3/ 4/ 11	0.9
6. Quinlan & Hunt Example 6			
501*nblit + 100*level		>500	
" - axiom 1 + commutativity	A	277/277/359/1803	190.7
" - axiom 2 + associativity		>500	
" - axioms 1&2 + commu. + assoc.	A	260/260/334/1431	165.3
nblit + size blits		>500	
" - axiom 1 + commutativity		>500	
" - axiom 2 + associativity		>500	
" - axioms 1&2 + commu. + assoc.		>500	
7. Quinlan & Hunt Example 7			
501*nblit + 100*level		>500	
" - axiom 1 + commutativity	A	21/ 21/ 26/137	11.7
" - axiom 2 + associativity	B	17/ 17/ 21/112	9.5
" - axioms 1&2 + commu. + assoc.	C	17/ 17/ 22/ 95	9.5
nblit + size blits	D	95/ 95/131/671	53.2
" - axiom 1 + commutativity	E	45/ 45/ 61/292	22.5
" - axiom 2 + associativity	F	74/ 74/106/452	43.6
" - axioms 1&2 + commu. + assoc.	C	8/ 8/ 10/ 42	3.6
8. Quinlan & Hunt Example 8 (Chang Example 10c)			
501 100 - -	A	2615/2615/3481/20145	615.9+
" - axiom 1 + commutativity	B	5/ 5/ 6/ 27	2.3
" - axiom 2 + associativity		>500	
" - axioms 1&2 + commu. + assoc.	B	5/ 5/ 6/ 24	1.5
nblit + size blits	C	285/285/407/2100	168.9
" - axiom 1 + commutativity	B	3/ 3/ 4/ 12	1.1
" - axiom 2 + associativity	D	21/ 21/ 30/134	7.5
" - axioms 1&2 + commu. + assoc.	B	3/ 3/ 4/ 11	0.9

Search Strategy Employed/ Options Used	Proof Code	Chains Generated (ret/acc/tot/att)	Search Time (sec)
9. Quinlan & Hunt Example 9 (Chang Example 10d)			
501 100 - -	A	48/ 48/ 63/364	11.4+
" - axiom 1 + commutativity	B	1/ 1/ 1/ 2	0.3
" - axiom 2 + associativity	A	59/ 59/ 76/387	32.7
" - axioms 1&2 + commu. + assoc.	B	1/ 1/ 1/ 2	0.3
nblit + size blits	A	12/ 12/ 17/ 83	5.2
" - axiom 1 + commutativity	B	1/ 1/ 1/ 2	0.5
" - axiom 2 + associativity	A	12/ 12/ 17/ 72	5.0
" - axioms 1&2 + commu. + assoc.	B	1/ 1/ 1/ 2	0.4
10. Quinlan & Hunt Example 10			
501*nblit + 100*level	A	30/ 30/ 38/228	14.2
" - axiom 1 + commutativity	A	34/ 34/ 44/224	17.2
" - axiom 2 + associativity	A	30/ 30/ 38/200	15.6
" - axioms 1&2 + commu. + assoc.	A	34/ 34/ 44/192	13.4
nblit + size blits	A	13/ 13/ 16/ 90	5.2
" - axiom 1 + commutativity	A	13/ 13/ 16/ 79	5.9
" - axiom 2 + associativity	A	13/ 13/ 16/ 79	5.4
" - axioms 1&2 + commu. + assoc.	A	13/ 13/ 16/ 68	4.6
11. Quinlan & Hunt Example 11			
501*nblit + 100*level		>500	
" - axiom 1 + commutativity	A	256/256/329/1673	172.5
" - axiom 2 + associativity	B	130/130/172/866	63.1
" - axioms 1&2 + commu. + assoc.	C	31/ 31/ 40/173	28.1
nblit + size blits	D	326/326/440/2335	172.6
" - axiom 1 + commutativity	E	18/ 18/ 22/118	7.1
" - axiom 2 + associativity	F	68/ 68/ 98/433	38.8
" - axioms 1&2 + commu. + assoc.	C	13/ 13/ 16/ 70	5.3
12. Quinlan & Hunt Example 18			
501*nblit + 100*level		>500	
" - axiom 1 + commutativity	A	316/316/405/2064	247.3
" - axiom 2 + associativity		>500	
" - axioms 1&2 + commu. + assoc.	A	292/292/372/1614	169.4
nblit + size blits		>500	
" - axiom 1 + commutativity		>500	
" - axiom 2 + associativity	B	490/490/763/3092	334.9
" - axioms 1&2 + commu. + assoc.	C	44/ 44/ 60/238	23.6

B.3. Proofs

1. Quinlan & Hunt Example 1 (Chang Example 10a)

Proof A

12.	(a+b)+c→a+(b+c)	theorem
13.	[(a+b)+c→a+(b+c)] (a+b)+c→x1 x1→a+(b+c)	extend by 7
14.	[(a+b)+c→a+(b+c)] (a+b)+c→(b+c)+a	extend by 1
15.	[(a+b)+c→a+(b+c)] [(a+b)+c→(b+c)+a] (a+b)+c→x1 x1→(b+c)+a	extend by 7
16.	[(a+b)+c→a+(b+c)] [(a+b)+c→(b+c)+a] (a+b)+c→b+(c+a)	extend by 2
17.	[(a+b)+c→a+(b+c)] [(a+b)+c→(b+c)+a] [(a+b)+c→b+(c+a)] (a+b)+c→x1 x1→b+(c+a)	extend by 7
18.	[(a+b)+c→a+(b+c)] [(a+b)+c→(b+c)+a] [(a+b)+c→b+(c+a)] (a+b)+c→(c+a)+b	extend by 1
19.	[(a+b)+c→a+(b+c)] [(a+b)+c→(b+c)+a] [(a+b)+c→b+(c+a)] [(a+b)+c→(c+a)+b] (a+b)+c→x1 x1→(c+a)+b	extend by 7
20.	[(a+b)+c→a+(b+c)] [(a+b)+c→(b+c)+a] [(a+b)+c→b+(c+a)] [(a+b)+c→(c+a)+b] (a+b)+c→c+(a+b)	extend by 2
21.	∅	extend by 1

Proof B

12.	(a+b)+c→a+(b+c)	theorem
13.	∅	extend by 2*

Proof C

12.	a+b+c→a+b+c	theorem
13.	∅	extend by 8**

2. Quinlan & Hunt Example 2

Proof A

12.	(a-b)+b→a	theorem
13.	[(a-b)+b→a] (a-b)+b→x1 x1→a	extend by 7
14.	[(a-b)+b→a] (a-b)+b→(a+x1)-x1	extend by 3
15.	∅	extend by 5

3. Quinlan & Hunt Example 3

Proof A

12.	a→(a-b)+b	theorem
13.	[a→(a-b)+b] a→x1 x1→(a-b)+b	extend by 7
14.	[a→(a-b)+b] a→(a+b)-b	extend by 6
15.	∅	extend by 4

4. Quinlan & Hunt Example 4

Proof A

12.	a+(b-c)→(a+b)-c	theorem
13.	[a+(b-c)→(a+b)-c] a+(b-c)→x1 x1→(a+b)-c	extend by 7
14.	[a+(b-c)→(a+b)-c] a+(b-c)→x1 [x1→(a+b)-c] x2→a+b x1→x2-c	extend by 10
15.	[a+(b-c)→(a+b)-c] a+(b-c)→(x1-c)+x2 [(x1-c)+x2→(a+b)-c] x1+x2→a+b	extend by 5
16.	[a+(b-c)→(a+b)-c] a+(b-c)→(b-c)+a	extend by 1
17.	∅	extend by 1

Proof B

12.	a+(b-c)→(a+b)-c	theorem

```
    13. ∅                                                                                                                    extend by 5*

Proof C
    12. a+(b-c)÷(a+b)-c                                                                                                      theorem
    13. [a+(b-c)÷(a+b)-c] a+(b-c)÷x1 x1÷(a+b)-c                                                                              extend by 7
    14. [a+(b-c)÷(a+b)-c] a+(b-c)÷(((a+b)-c)+x1)-x1                                                                          extend by 3
    15. [a+(b-c)÷(a+b)-c] [a+(b-c)÷(((a+b)-c)+x1)-x1]  x2÷((a+b)-c)+x1 a+(b-c)÷x2-x1                                         extend by 10
    16. [a+(b-c)÷(a+b)-c] [a+(b-c)÷(((a+b)-c)+x1)-x1] a+(b-c)+x1÷((a+b)-c)+x1                                                extend by 4
    17. [a+(b-c)÷(a+b)-c] [a+(b-c)÷(((a+b)-c)+x1)-x1] [a+(b-c)+x1÷((a+b)-c)+x1] x2÷(a+b)-c
        a+(b-c)+x1÷x2+x1                                                                                                     extend by 9
    18. [a+(b-c)÷(a+b)-c] [a+(b-c)÷(((a+b)-c)+a)-a] [a+(b-c)+a÷((a+b)-c)+a] (b-c)+a÷(a+b)-c                                  extend by 1**
    19. [a+(b-c)÷(a+b)-c] [a+(b-c)÷(((a+b)-c)+a)-a] [a+(b-c)+a÷((a+b)-c)+a] [(b-c)+a÷(a+b)-c]
        x1÷a+b (b-c)+a÷x1-c                                                                                                  extend by 10
    20. [a+(b-c)÷(a+b)-c] [a+(b-c)÷(((a+b)-c)+a)-a] [a+(b-c)+a÷((a+b)-c)+a] [(b-c)+a÷(a+b)-c] b+a÷a+b                       extend by 5
    21. ∅                                                                                                                    extend by 1

Proof D
    12. a+(b-c)÷(a+b)-c                                                                                                      theorem
    13. [a+(b-c)÷(a+b)-c] x1÷a+b a+(b-c)÷x1-c                                                                                extend by 10
    14. [a+(b-c)÷(a+b)-c] a+(b-c)+c÷a+b                                                                                      extend by 4
    15. [a+(b-c)÷(a+b)-c] [a+(b-c)+c÷a+b] a+(b-c)+c÷x1 x1÷a+b                                                                extend by 7
    16. [a+(b-c)÷(a+b)-c] [a+(b-c)+c÷a+b] a+(b-c)+c÷b+a                                                                      extend by 1
    17. [a+(b-c)÷(a+b)-c] [a+(b-c)+c÷a+b] [a+(b-c)+c÷b+a] [a+(b-c)+c÷b+a] x1÷b a+(b-c)+c÷x1+a                               extend by 9
    18. [a+(b-c)÷(a+b)-c] [a+(b-c)+c÷a+b] [a+(b-c)+c÷b+a] [a+(b-c)+c÷b+a] (b-c)+c÷b                                         extend by 1**
    19. [a+(b-c)÷(a+b)-c] [a+(b-c)+c÷a+b] [a+(b-c)+c÷b+a] [a+(b-c)+c÷b+a] [(b-c)+c÷b] (b-c)+c÷x1
        x1÷b                                                                                                                 extend by 7
    20. [a+(b-c)÷(a+b)-c] [a+(b-c)+c÷a+b] [a+(b-c)+c÷b+a] [a+(b-c)+c÷b+a] [(b-c)+c÷b]
        (b-c)+c÷(b+x1)-x1                                                                                                    extend by 3
    21. ∅                                                                                                                    extend by 5

5. Quinlan & Hunt Example 5 (Chang Example 10b)

Proof A
    12. (a-b)+c÷a+(c-b)                                                                                                      theorem
    13. [(a-b)+c÷a+(c-b)] (a-b)+c÷x1 x1÷a+(c-b)                                                                              extend by 7
    14. [(a-b)+c÷a+(c-b)] (a-b)+c÷(c-b)+a                                                                                    extend by 1
    15. [(a-b)+c÷a+(c-b)] [(a-b)+c÷(c-b)+a] (a-b)+c÷x1 x1÷(c-b)+a                                                            extend by 7
    16. [(a-b)+c÷a+(c-b)] [(a-b)+c÷(c-b)+a] (a-b)+c÷(c+a)-b                                                                  extend by 6
    17. [(a-b)+c÷a+(c-b)] [(a-b)+c÷(c-b)+a] [(a-b)+c÷(c+a)-b] x1÷c+a (a-b)+c÷x1-b                                           extend by 10
    18. [(a-b)+c÷a+(c-b)] [(a-b)+c÷(c-b)+a] [(a-b)+c÷(c+a)-b] a+c÷c+a                                                       extend by 5
    19. ∅                                                                                                                    extend by 1

Proof B
    12. (a-b)+c÷a+(c-b)                                                                                                      theorem
    13. [(a-b)+c÷a+(c-b)] (a-b)+c÷x1 x1÷a+(c-b)                                                                              extend by 7
    14. [(a-b)+c÷a+(c-b)] (a-b)+c÷(c+a)-b                                                                                    extend by 6*
    15. ∅                                                                                                                    extend by 5*

6. Quinlan & Hunt Example 6

Proof A
    12. (a-c)-(b-c)÷a-b                                                                                                      theorem
    13. [(a-c)-(b-c)÷a-b] x1÷a (a-c)-(b-c)÷x1-b                                                                              extend by 10
    14. [(a-c)-(b-c)÷a-b] ((a-c)-(b-c))+b÷a                                                                                  extend by 4
```

```
    15. [(a-c)-(b-c)→a-b] [((a-c)-(b-c))+b→a] ((a-c)-(b-c))+b→x1 x1→a                                   extend by 7
    16. [(a-c)-(b-c)→a-b] [((a-c)-(b-c))+b→a] ((a-c)-(b-c))+b→(a+x1)-x1                                extend by 3
    17. [(a-c)-(b-c)→a-b] [((a-c)-(b-c))+b→a] [((a-c)-(b-c))+b→(a+x1)-x1] x2→a+x1
        ((a-c)-(b-c))+b→x2-x1                                                                        extend by 10
    18. [(a-c)-(b-c)→a-b] [((a-c)-(b-c))+b→a] [((a-c)-(b-c))+b→(a+(b-c))-(b-c)] (a-c)+b→a+(b-c)      extend by 5
    19. [(a-c)-(b-c)→a-b] [((a-c)-(b-c))+b→a] [((a-c)-(b-c))+b→(a+(b-c))-(b-c)] [(a-c)+b→a+(b-c)]
        (a-c)+b→x1 x1→a+(b-c)                                                                        extend by 7
    20. [(a-c)-(b-c)→a-b] [((a-c)-(b-c))+b→a] [((a-c)-(b-c))+b→(a+(b-c))-(b-c)] [(a-c)+b→a+(b-c)]
        (a-c)+b→(b+a)-c                                                                              extend by 6*
    21. ∅                                                                                            extend by 5*
```

7. Quinlan & Hunt Example 7

Proof A

```
    12. (a+c)-(b+c)→a-b                                                                              theorem
    13. [(a+c)-(b+c)→a-b] x1→a (a+c)-(b+c)→x1-b                                                      extend by 10
    14. [(a+c)-(b+c)→a-b] ((a+c)-(b+c))+b→a                                                          extend by 4
    15. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] ((a+c)-(b+c))+b→x1 x1→a                                extend by 7
    16. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] ((a+c)-(b+c))+b→(a+x1)-x1                              extend by 3
    17. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] [((a+c)-(b+c))+b→(a+x1)-x1] x2→a+x1
        ((a+c)-(b+c))+b→x2-x1                                                                        extend by 10
    18. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] [((a+c)-(b+c))+b→(a+(b+c))-(b+c)] (a+c)+b→a+(b+c)      extend by 5
    19. ∅                                                                                            extend by 2*
```

Proof B

```
    12. (a+c)-(b+c)→a-b                                                                              theorem
    13. [(a+c)-(b+c)→a-b] x1→a (a+c)-(b+c)→x1-b                                                      extend by 10
    14. [(a+c)-(b+c)→a-b] ((a+c)-(b+c))+b→a                                                          extend by 4
    15. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] ((a+c)-(b+c))+b→x1 x1→a                                extend by 7
    16. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] ((a+c)-(b+c))+b→(a+x1)-x1                              extend by 3
    17. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] [((a+c)-(b+c))+b→(a+x1)-x1] x2→x1
        ((a+c)-(b+c))+b→(a+x1)-x2                                                                    extend by 11
    18. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] [((a+c)-(b+c))+b→(a+c+b)-(c+b)] b+c→c+b                 extend by 5**
    19. ∅                                                                                            extend by 1
```

Proof C

```
    12. (a+c)-(b+c)→a-b                                                                              theorem
    13. [(a+c)-(b+c)→a-b] x1→a (a+c)-(b+c)→x1-b                                                      extend by 10
    14. [(a+c)-(b+c)→a-b] ((a+c)-(b+c))+b→a                                                          extend by 4
    15. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] ((a+c)-(b+c))+b→x1 x1→a                                extend by 7
    16. [(a+c)-(b+c)→a-b] [((a+c)-(b+c))+b→a] ((a+c)-(b+c))+b→(a+x1)-x1                              extend by 3
    17. ∅                                                                                            extend by 5***
```

Proof D

```
    12. (a+c)-(b+c)→a-b                                                                              theorem
    13. [(a+c)-(b+c)→a-b] x1→a (a+c)-(b+c)→x1-b                                                      extend by 10
    14. [(a+c)-(b+c)→a-b] x1→a [(a+c)-(b+c)→x1-b] (a+c)-(b+c)→x2 x2→x1-b                             extend by 7
    15. [(a+c)-(b+c)→a-b] x1+b→a [(a+c)-(b+c)→(x1+b)-b] (a+c)-(b+c)→x1                               extend by 4
    16. [(a+c)-(b+c)→a-b] x1+b→a [(a+c)-(b+c)→(x1+b)-b] [(a+c)-(b+c)→x1] (a+c)-(b+c)→x2 x2→x1       extend by 7
    17. [(a+c)-(b+c)→a-b] (x1+x2)+b→a [(a+c)-(b+c)→((x1+x2)+b)-b] [(a+c)-(b+c)→x1+x2]
        (a+c)-(b+c)→x2+x1                                                                            extend by 1
    18. [(a+c)-(b+c)→a-b] (c+(a-(b+c)))+b→a                                                          extend by 6
    19. [(a+c)-(b+c)→a-b] [(c+(a-(b+c)))+b→a] (c+(a-(b+c)))+b→x1 x1→a                                extend by 7
    20. [(a+c)-(b+c)→a-b] [(c+(a-(b+c)))+b→a] (c+(a-(b+c)))+b→x1 [x1→a] x1→x2 x2→a                   extend by 7
```

```
21. [(a+c)-(b+c)+a-b] [(c+(a-(b+c)))+b+a] (c+(a-(b+c)))+b+x1  [x1+a] x1+(a+x2)-x2                    extend by 3
22. [(a+c)-(b+c)+a-b] [(c+(a-(b+c)))+b+a] (c+(a-(b+c)))+b+(a-x1)+x1                                    extend by 5
23. [(a+c)-(b+c)+a-b] [(c+(a-(b+c)))+b+a] [(c+(a-(b+c)))+b+(a-x1)+x1] (c+(a-(b+c)))+b+x2
    x2+(a-x1)+x1                                                                                     extend by 7
24. [(a+c)-(b+c)+a-b] [(c+(a-(b+c)))+b+a] [(c+(a-(b+c)))+b+(a-x1)+x1] (c+(a-(b+c)))+b+x1+(a-x1)     extend by 1
25. [(a+c)-(b+c)+a-b] [(c+(a-(b+c)))+b+a] [(c+(a-(b+c)))+b+(a-x1)+x1] [(c+(a-(b+c)))+b+x1+(a-x1)]
    (c+(a-(b+c)))+b+x2 x2+x1+(a-x1)                                                                  extend by 7
26. [(a+c)-(b+c)+a-b] [(c+(a-(b+c)))+b+a] [(c+(a-(b+c)))+b+(a-(x1+x2))+(x1+x2)]
    [(c+(a-(b+c)))+b+(x1+x2)+(a-(x1+x2))] (c+(a-(b+c)))+b+x1+(x2+(a-(x1+x2)))                        extend by 2
27. ∅                                                                                                extend by 1
```

Proof E

```
12. (a+c)-(b+c)+a-b                                                                                  theorem
13. [(a+c)-(b+c)+a-b] x1+a (a+c)-(b+c)+x1-b                                                          extend by 10
14. [(a+c)-(b+c)+a-b] x1+a [(a+c)-(b+c)+x1-b] (a+c)-(b+c)+x2 x2+x1-b                                 extend by 7
15. [(a+c)-(b+c)+a-b] x1+b+a [(a+c)-(b+c)+(x1+b)-b] (a+c)-(b+c)+x1                                   extend by 4
16. [(a+c)-(b+c)+a-b] ((a-(b+c))+c)+b+a                                                              extend by 6
17. [(a+c)-(b+c)+a-b] [((a-(b+c))+c)+b+a] ((a-(b+c))+c)+b+x1 x1+a                                    extend by 7
18. [(a+c)-(b+c)+a-b] [((a-(b+c))+c)+b+a] ((a-(b+c))+c)+b+x1 [x1+a] x1+x2 x2+a                       extend by 7
19. [(a+c)-(b+c)+a-b] [((a-(b+c))+c)+b+a] ((a-(b+c))+c)+b+x1 [x1+a] x1+(a+x2)-x2                     extend by 3
20. [(a+c)-(b+c)+a-b] [((a-(b+c))+c)+b+a] ((a-(b+c))+c)+b+(a-x1)+x1                                  extend by 5
21. ∅                                                                                                extend by 2*
```

Proof F

```
12. (a+c)-(b+c)+a-b                                                                                  theorem
13. [(a+c)-(b+c)+a-b] x1+a (a+c)-(b+c)+x1-b                                                          extend by 10
14. [(a+c)-(b+c)+a-b] x1+a [(a+c)-(b+c)+x1-b] (a+c)-(b+c)+x2 x2+x1-b                                 extend by 7
15. [(a+c)-(b+c)+a-b] x1+b+a [(a+c)-(b+c)+(x1+b)-b] (a+c)-(b+c)+x1                                   extend by 4
16. [(a+c)-(b+c)+a-b] x1+b+a [(a+c)-(b+c)+(x1+b)-b] [(a+c)-(b+c)+x1] (a+c)-(b+c)+x2 x2+x1           extend by 7
17. [(a+c)-(b+c)+a-b] x1+x2+b+a [(a+c)-(b+c)+(x1+x2+b)-b] [(a+c)-(b+c)+x1+x2] (a+c)-(b+c)+x2+x1     extend by 1
18. [(a+c)-(b+c)+a-b] c+(a-(b+c))+b+a                                                                extend by 6
19. [(a+c)-(b+c)+a-b] [c+(a-(b+c))+b+a] c+(a-(b+c))+b+x1 x1+a                                        extend by 7
20. [(a+c)-(b+c)+a-b] [c+(a-(b+c))+b+a] c+(a-(b+c))+b+x1 [x1+a] x1+x2 x2+a                           extend by 7
21. [(a+c)-(b+c)+a-b] [c+(a-(b+c))+b+a] c+(a-(b+c))+b+x1 [x1+a] x1+(a+x2)-x2                         extend by 3
22. [(a+c)-(b+c)+a-b] [c+(a-(b+c))+b+a] c+(a-(b+c))+b+(a-x1)+x1                                      extend by 5
23. ∅                                                                                                extend by 1**
```

8. Quinlan & Hunt Example 8 (Chang Example 10c)

Proof A

```
12. a+(b-c)+(a-c)+b                                                                                  theorem
13. [a+(b-c)+(a-c)+b] a+(b-c)+x1 x1+(a-c)+b                                                          extend by 7
14. [a+(b-c)+(a-c)+b] a+(b-c)+(a+b)-c                                                                extend by 6
15. [a+(b-c)+(a-c)+b] [a+(b-c)+(a+b)-c] a+(b-c)+x1 x1+(a+b)-c                                        extend by 7
16. [a+(b-c)+(a-c)+b] [a+(b-c)+(a+b)-c] a+(b-c)+x1 [x1+(a+b)-c] x2+a+b x1+x2-c                       extend by 10
17. [a+(b-c)+(a-c)+b] [a+(b-c)+(a+b)-c] a+(b-c)+(x1-c)+x2 [(x1-c)+x2+(a+b)-c] x1+x2+a+b             extend by 5
18. [a+(b-c)+(a-c)+b] [a+(b-c)+(a+b)-c] a+(b-c)+(b-c)+a                                              extend by 1
19. ∅                                                                                                extend by 1
```

Proof B

```
12. a+(b-c)+(a-c)+b                                                                                  theorem
13. [a+(b-c)+(a-c)+b] a+(b-c)+x1 x1+(a-c)+b                                                          extend by 7
14. [a+(b-c)+(a-c)+b] a+(b-c)+(a+b)-c                                                                extend by 6
15. ∅                                                                                                extend by 5*
```

Proof C

12.	a+(b-c)→(a-c)+b	theorem
13.	[a+(b-c)→(a-c)+b] a+(b-c)→x1 x1→(a-c)+b	extend by 7
14.	[a+(b-c)→(a-c)+b] a+(b-c)→(a+b)-c	extend by 6
15.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] x1→a+b a+(b-c)→x1-c	extend by 10
16.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] x1→a+b [a+(b-c)→x1-c] a+(b-c)→x2 x2→x1-c	extend by 7
17.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] x1+x2→a+b [a+(b-c)→(x1+x2)-c] a+(b-c)→(x1-c)+x2	extend by 5
18.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] b+a→a+b	extend by 1
19.	∅	extend by 1

Proof D

12.	a+(b-c)→(a-c)+b	theorem
13.	[a+(b-c)→(a-c)+b] a+(b-c)→x1 x1→(a-c)+b	extend by 7
14.	[a+(b-c)→(a-c)+b] a+(b-c)→(a+b)-c	extend by 6
15.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] x1→a+b a+(b-c)→x1-c	extend by 10
16.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] a+(b-c)+c→a+b	extend by 4
17.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] [a+(b-c)+c→a+b] a+(b-c)+c→x1 x1→a+b	extend by 7
18.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] [a+(b-c)+c→a+b] a+(b-c)+c→b+a	extend by 1
19.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] [a+(b-c)+c→a+b] [a+(b-c)+c→b+a] x1→b a+(b-c)+c→x1+a	extend by 9
20.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] [a+(b-c)+c→a+b] [a+(b-c)+c→b+a] (b-c)+c→b	extend by 1**
21.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] [a+(b-c)+c→a+b] [a+(b-c)+c→b+a] [(b-c)+c→b] (b-c)+c→x1 x1→b	extend by 7
22.	[a+(b-c)→(a-c)+b] [a+(b-c)→(a+b)-c] [a+(b-c)+c→a+b] [a+(b-c)+c→b+a] [(b-c)+c→b] (b-c)+c→(b+x1)-x1	extend by 3
23.	∅	extend by 5

9. Quinlan & Hunt Example 9 (Chang Example 10d)

Proof A

12.	(a+b)-c→a+(b-c)	theorem
13.	[(a+b)-c→a+(b-c)] (a+b)-c→x1 x1→a+(b-c)	extend by 7
14.	[(a+b)-c→a+(b-c)] (a+b)-c→(b-c)+a	extend by 1
15.	[(a+b)-c→a+(b-c)] [(a+b)-c→(b-c)+a] (a+b)-c→x1 x1→(b-c)+a	extend by 7
16.	[(a+b)-c→a+(b-c)] [(a+b)-c→(b-c)+a] (a+b)-c→(b+a)-c	extend by 6
17.	[(a+b)-c→a+(b-c)] [(a+b)-c→(b-c)+a] [(a+b)-c→(b+a)-c] x1→b+a (a+b)-c→x1-c	extend by 10
18.	[(a+b)-c→a+(b-c)] [(a+b)-c→(b-c)+a] [(a+b)-c→(b+a)-c] a+b→b+a	extend by 8
19.	∅	extend by 1

Proof B

12.	(a+b)-c→a+(b-c)	theorem
13.	∅	extend by 6*

10. Quinlan & Hunt Example 10

Proof A

12.	(a-b)-c→(a-c)-b	theorem
13.	[(a-b)-c→(a-c)-b] x1→a-c (a-b)-c→x1-b	extend by 10
14.	[(a-b)-c→(a-c)-b] [(a-b)-c)+b→a-c	extend by 4
15.	[(a-b)-c→(a-c)-b] [((a-b)-c)+b→a-c] x1→a ((a-b)-c)+b→x1-c	extend by 10
16.	[(a-b)-c→(a-c)-b] [((a-b)-c)+b→a-c] (a-b)+b→a	extend by 5
17.	[(a-b)-c→(a-c)-b] [((a-b)-c)+b→a-c] [(a-b)+b→a] (a-b)+b→x1 x1→a	extend by 7
18.	[(a-b)-c→(a-c)-b] [((a-b)-c)+b→a-c] [(a-b)+b→a] (a-b)+b→(a+x1)-x1	extend by 3
19.	∅	extend by 5

11. Quinlan & Hunt Example 11

Proof A

12.	a-(b+c)+(a-b)-c	theorem
13.	[a-(b+c)+(a-b)-c] x1+a-b a-(b+c)+x1-c	extend by 10
14.	[a-(b+c)+(a-b)-c] (a-(b+c))+c+a-b	extend by 4
15.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+a (a-(b+c))+c+x1-b	extend by 10
16.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c)+b+a]	extend by 4
17.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [[(a-(b+c))+c)+b+a] ((a-(b+c))+c)+b+x1 x1+a	extend by 7
18.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [[(a-(b+c))+c)+b+a] ((a-(b+c))+c)+b+(a+x1)-x1	extend by 3
19.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [[(a-(b+c))+c)+b+a] [((a-(b+c))+c)+b+(a+x1)-x1] ((a-(b+c))+c)+b+x2 x2+(a+x1)-x1	extend by 7
20.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [[(a-(b+c))+c)+b+a] [((a-(b+c))+c)+b+(a+x1)-x1] ((a-(b+c))+c)+b+(a-x1)+x1	extend by 5
21.	∅	extend by 2*

Proof B

12.	a-(b+c)+(a-b)-c	theorem
13.	[a-(b+c)+(a-b)-c] x1+a-b a-(b+c)+x1-c	extend by 10
14.	[a-(b+c)+(a-b)-c] (a-(b+c))+c+a-b	extend by 4
15.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+a (a-(b+c))+c+x1-b	extend by 10
16.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] (a-(b+c))+c+b+a	extend by 4
17.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c+b+a] (a-(b+c))+c+b+x1 x1+a	extend by 7
18.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c+b+a] (a-(b+c))+c+b+(a+x1)-x1	extend by 3
19.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c+b+a] [(a-(b+c))+c+b+(a+x1)-x1] x2+x1 (a-(b+c))+c+b+(a+x1)-x2	extend by 11
20.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c+b+a] [(a-(b+c))+c+b+(a+c+b)-(c+b)] b+c+c+b	extend by 5**
21.	∅	extend by 1

Proof C

12.	a-(b+c)+(a-b)-c	theorem
13.	[a-(b+c)+(a-b)-c] x1+a-b a-(b+c)+x1-c	extend by 10
14.	[a-(b+c)+(a-b)-c] (a-(b+c))+c+a-b	extend by 4
15.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+a (a-(b+c))+c+x1-b	extend by 10
16.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] (a-(b+c))+c+b+a	extend by 4
17.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c+b+a] (a-(b+c))+c+b+x1 x1+a	extend by 7
18.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] [(a-(b+c))+c+b+a] (a-(b+c))+c+b+(a+x1)-x1	extend by 3
19.	∅	extend by 5***

Proof D

12.	a-(b+c)+(a-b)-c	theorem
13.	[a-(b+c)+(a-b)-c] x1+a-b a-(b+c)+x1-c	extend by 10
14.	[a-(b+c)+(a-b)-c] x1+a-b [a-(b+c)+x1-c] a-(b+c)+x2 x2+x1-c	extend by 7
15.	[a-(b+c)+(a-b)-c] x1+c+a-b [a-(b+c)+(x1+c)-c] a-(b+c)+x1	extend by 4
16.	[a-(b+c)+(a-b)-c] x1+c+a-b [a-(b+c)+(x1+c)-c] [a-(b+c)+x1] a-(b+c)+x2 x2+x1	extend by 7
17.	[a-(b+c)+(a-b)-c] x1+c+a-b [a-(b+c)+(x1+c)-c] [a-(b+c)+x1] a-(b+c)+(x1+x2)-x2	extend by 3
18.	[a-(b+c)+(a-b)-c] (a-(b+c))+c+a-b	extend by 4
19.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+a (a-(b+c))+c+x1-b	extend by 10
20.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+a [(a-(b+c))+c+x1-b] (a-(b+c))+c+x2 x2+x1-b	extend by 7
21.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+b+a [(a-(b+c))+c+(x1+b)-b] (a-(b+c))+c+x1	extend by 4
22.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] x1+b+a [(a-(b+c))+c+(x1+b)-b] [(a-(b+c))+c+x1] (a-(b+c))+c+x2 x2+x1	extend by 7
23.	[a-(b+c)+(a-b)-c] [(a-(b+c))+c+a-b] (x1+x2)+b+a [(a-(b+c))+c+((x1+x2)+b)-b]	

```
      [(a-(b+c))+c+x1+x2] (a-(b+c))+c+x2+x1                                                
```

18. [(a+b)-c+a-(c-b)] [((a+b)-c)+(c-b)+a] [((a+b)-c)+(c-b)+(a+b)-b] c+((a+b)-c)+a+b — extend by 5*
19. [(a+b)-c+a-(c-b)] [((a+b)-c)+(c-b)+a] [((a+b)-c)+(c-b)+(a+b)-b] [c+((a+b)-c)+a+b] c+((a+b)-c)+x1 x1+a+b — extend by 7
20. [(a+b)-c+a-(c-b)] [((a+b)-c)+(c-b)+a] [((a+b)-c)+(c-b)+(a+b)-b] [c+((a+b)-c)+a+b] c+((a+b)-c)+((a+b)+x1)-x1 — extend by 3
21. ∅ — extend by 5*

Proof B

12. (a+b)-c+a-(c-b) — theorem
13. [(a+b)-c+a-(c-b)] x1+a (a+b)-c+x1-(c-b) — extend by 10
14. [(a+b)-c+a-(c-b)] x1+a [(a+b)-c+x1-(c-b)] (a+b)-c+x2 x2+x1-(c-b) — extend by 7
15. [(a+b)-c+a-(c-b)] x1+(c-b)+a [(a+b)-c+(x1+(c-b))-(c-b)] (a+b)-c+x1 — extend by 4
16. [(a+b)-c+a-(c-b)] (a-c)+b+(c-b)+a — extend by 6
17. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 x1+a — extend by 7
18. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 [x1+a] x1+x2 x2+a — extend by 7
19. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 [x1+a] x1+x2 [x2+a] x2+x3 x3+a — extend by 7
20. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 [x1+a] x1+x2 [x2+a] x2+(a+x3)-x3 — extend by 3
21. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 [x1+a] x1+(((a+x2)-x2)+x3)-x3 — extend by 3
22. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+(a+x1)-x1 — extend by 4
23. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+x1)-x1] (a-c)+b+(c-b)+x2 x2+(a+x1)-x1 — extend by 7
24. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+x1)-x1] (a-c)+b+(c-b)+(a-x1)+x1 — extend by 5
25. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+x1)-x1] [(a-c)+b+(c-b)+(a-x1)+x1] (a-c)+b+(c-b)+x2 x2+(a-x1)+x1 — extend by 7
26. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+x1)-x1] [(a-c)+b+(c-b)+(a-x1)+x1] (a-c)+b+(c-b)+x1+(a-x1) — extend by 1
27. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+x1)-x1] [(a-c)+b+(c-b)+(a-x1)+x1] [(a-c)+b+(c-b)+x1+(a-x1)] x2+x1 (a-c)+b+(c-b)+x2+(a-x1) — extend by 9
28. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+c)-c] [(a-c)+b+(c-b)+(a-c)+c] [(a-c)+b+(c-b)+c+(a-c)] b+(c-b)+c — extend by 1**
29. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+c)-c] [(a-c)+b+(c-b)+(a-c)+c] [(a-c)+b+(c-b)+c+(a-c)] [b+(c-b)+c] b+(c-b)+x1 x1+c — extend by 7
30. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+c)-c] [(a-c)+b+(c-b)+(a-c)+c] [(a-c)+b+(c-b)+c+(a-c)] [b+(c-b)+c] b+(c-b)+x1 [x1+c] x1+x2 x2+c — extend by 7
31. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+c)-c] [(a-c)+b+(c-b)+(a-c)+c] [(a-c)+b+(c-b)+c+(a-c)] [b+(c-b)+c] b+(c-b)+x1 [x1+c] x1+(c+x2)-x2 — extend by 3
32. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a+c)-c] [(a-c)+b+(c-b)+(a-c)+c] [(a-c)+b+(c-b)+c+(a-c)] [b+(c-b)+c] b+(c-b)+(c-x1)+x1 — extend by 5
33. ∅ — extend by 1

Proof C

12. (a+b)-c+a-(c-b) — theorem
13. [(a+b)-c+a-(c-b)] x1+a (a+b)-c+x1-(c-b) — extend by 10
14. [(a+b)-c+a-(c-b)] x1+a [(a+b)-c+x1-(c-b)] (a+b)-c+x2 x2+x1-(c-b) — extend by 7
15. [(a+b)-c+a-(c-b)] x1+(c-b)+a [(a+b)-c+(x1+(c-b))-(c-b)] (a+b)-c+x1 — extend by 4
16. [(a+b)-c+a-(c-b)] (a-c)+b+(c-b)+a — extend by 6
17. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 x1+a — extend by 7
18. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 [x1+a] x1+x2 x2+a — extend by 7
19. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+x1 [x1+a] x1+(a+x2)-x2 — extend by 3
20. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] (a-c)+b+(c-b)+(a-x1)+x1 — extend by 5
21. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a-x1)+x1] (a-c)+b+(c-b)+x2 x2+(a-x1)+x1 — extend by 7
22. [(a+b)-c+a-(c-b)] [(a-c)+b+(c-b)+a] [(a-c)+b+(c-b)+(a-x1)+x1] (a-c)+b+(c-b)+((a-x1)+x1+x2)-x2 — extend by 3
23. ∅ — extend by 5***

* operation requires commutativity
** operation requires associativity
*** operation requires both commutativity and associativity

References

1 Aubin, R. Some experimental results on SL-resolution. Memo No 66, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Edinburgh, July 1973.

2 Aubin, R. Personal communication. December 1973.

3 Belnap, N. D. Jr. How a computer should think. Contemporary Aspects of Philosophy. Proceedings of the Oxford International Symposium, 1975.

4 Belnap, N. D. Jr. A useful four-valued logic. Modern Uses of Multiple-valued Logic. Proceedings of the 1975 International Symposium on Multiple-valued Logic.

5 Bledsoe, W. W. Splitting and reduction heuristics in automatic theorem proving. Artificial Intelligence 2 (1971), 55-77.

8 Bledsoe, W. W., Boyer, R. S. and Henneman. Computer proofs of limit theorems. Artificial Intelligence 3 (1972), 27-60.

7 Bobrow, D. G. and Raphael, B. New programming languages for artificial intelligence research. Computing Surveys 6,3 (September 1974), 153-174.

8 Bobrow, R. J., Burton, R. R., Jacobs, J. M. and Lewis, D. UCI LISP Manual. University of California, Irvine, Calif., 1973.

9 Chang, C. L. The unit proof and the input proof in theorem proving. J. ACM 17,4 (October 1970), 698-707.

10 Davies, D. J. M. Popler 1.5 reference manual. TPU Report 1, University of Edinburgh, May 1973.

11 Fikes, R. E. and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving Artificial Intelligence 2 (1971), 189-208.

12 Fleisig, S., Loveland, D., Smiley, A. K. III and Yarmush, D. L. An implementation of the model elimination proof procedure. J. ACM 21,1 (January 1974), 124-139.

13 Gelernter, H. Machine generated problem solving graphs. Proceedings of Symposium on Mathematical Theory of Automata. Polytechnic Press, Brooklyn, 1963, 179-203.

14 Gelernter, H. Realization of a geometry-theorem proving machine. In Feigenbaum, E. A. and Feldman, J. (eds.). Computers and Thought. McGraw-Hill, New York, 1963, 134-152.

15 Gelernter, H., Hansen, J. R. and Loveland, D. W. Empirical explorations of the geometry-theorem proving machine. In Feigenbaum, E. A. and Feldman, J. (eds.). Computers and Thought. McGraw-Hill, New York, 1963, 153-163.

16 Gilmore, P. C. An examination of the geometry theorem machine. Artificial Intelligence 1 (1970), 171-187.

17 Green, C. C. Application of theorem proving to problem solving. Proceedings of International Joint Conference on Artificial Intelligence, Washington, D.C., 1969.

18 Henschen, L. J. Semantic resolution for Horn sets. Advance Papers of Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., 1975, pp. 46-52.

19 Henschen, L. and Wos, L. A theorem-proving language for experimentation. Comm. ACM 17,6 (June 1974), 308-314.

20 Henschen, L. and Wos, L. Unit refutations and Horn sets. J. ACM 21,4 (October 1974), 590-605.

21 Hewitt, C. Description and theoretical analysis using schemata of PLANNER: a language for proving theorems and manipulating models in a robot. Artificial Intelligence Laboratory report AI TR-258, Massachusetts Institute of Technology, Cambridge, Mass., April 1972.

22 Hewitt, C. How to use what you know. Advance Papers of Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., 1975, pp. 189-198.

23 Hewitt, C., Bishop, P. and Steiger, R. A universal ACTOR formalism for artificial intelligence. Advance Papers of Third International Joint Conference on Artificial Intelligence, Stanford, Calif., 1973, pp. 235-245.

24 Knuth, D. E. and Bendix, P. B. Simple word problems in universal algebras. In Leech, J. (ed.). Computational Problems in Abstract Algebras. Pergammon Press, 1970, pp. 263-297.

25 Kowalski, R. and Kuehner, D. Linear resolution with selection function. Artificial Intelligence 2 (1971), 227-260.

26 Lankford, D. S. Canonical algebraic simplification in computational logic. Technical Report, Mathematics Department, University of Texas at Austin, May 1975.

27 Lankford, D. S. Canonical inference. Technical Report, Mathematics Department, University of Texas at Austin, December 1975.

28 Lee, R. C. T. Fuzzy logic and the resolution principle. J. ACM 19,1 (January 1972), 109-119.

29 Loveland, D. W. A simplified format for the model elimination theorem-proving procedure. J. ACM 16,3 (July 1969), 349-363.

30 Loveland, D. W. A unifying view of some linear Herbrand procedures. J. ACM 19,2 (April 1972), 366-384.

31 Loveland, D. W. Forthcoming book. North-Holland, Amsterdam.

32 Loveland, D. W. and Stickel, M. E. The hole in goal trees: some guidance from resolution theory. Advance Papers of Third International Joint Conference on Artificial Intelligence, Stanford, Calif., 1973, pp. 153-161.

33 McDermott, D. V. and Sussman, G. J. The CONNIVER reference manual. AI Memo 259, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., May 1972.

34 Nevins, A. J. A human oriented logic for automatic theorem proving. J. ACM 21,4 (October 1974), 606-621.

35 Nilsson, N. J. Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York, 1971.

36 Reboh, R., Raphael, B., Yates, R. A., Kling, R. E. and Velarde, C. Study of automatic theorem-proving programs. Technical Note 75, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., November 1972.

37 Reboh, R. and Sacerdoti, E. A preliminary QLISP manual. Technical Note 81, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., August 1973.

38 Reiter, R. The use of models in automatic theorem-proving. University of British Columbia Technical Report 72-09 (September 1972).

39 Reiter, R. A semantically guided deductive system for automatic theorem-proving. Advance Papers of Third International Joint Conference on Artificial Intelligence. Stanford, Calif., 1973, pp. 41-50.

40 Rescher, N. Hypothetical Reasoning. North-Holland, Amsterdam, 1964.

41 Robinson, J. A. A machine-oriented logic based on the resolution principle. J. ACM 12,1 (January 1965), 23-41.

42 Robinson, J. A. A review of automatic theorem proving. Proceedings of Symposium on Applied Mathematics 19 (1966), 1-18.

43 Rulifson, J. F., Derksen, J. A. and Waldinger, R. J. QA4: a procedural calculus for intuitive reasoning. Technical Note 73, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., November 1972.

44 Sacerdoti, E., Fikes, R., Reboh, R., Sagalowics, D., Waldinger, R. and Wilber, M. QLISP: a language for the interactive development of complex systems. Technical Note 120, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., March 1976.

45 Shostak, R. E. Refutation graphs. Artificial Intelligence 7 (1976), 51-64.

46 Slagle, J. R. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. J. ACM 21,4 (October 1974), 622-642.

47 Stickel, M. E. The programmable strategy theorem prover: an implementation of the linear MESON procedure. Research Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, June 1974.

48 Stickel, M. E. A complete unification algorithm for associative-commutative functions. Advance Papers of Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., 1975, pp. 71-76.

49 Stickel, M. E. and Peterson, G. E. Complete sets of reductions for equational theories with complete unification algorithms. Technical Report, Department of Computer Science, University of Arizona, Tucson, September 1977.

50 Tarski, A. A decision method for elementary algebra and geometry. University of California Press, Berkeley, 1951.

51 Waldinger, R. J. Constructing Programs Automatically Using Theorem Proving. Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, 1969.

52 Wang, H. Toward mechanical mathematics. IBM J. of Research and Development 4,1 (January 1960), 2-22.

53 Wilber, M. A QLISP reference manual. Technical Note 118, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Calif., March 1976.

54 Wos, L., Robinson, G. A., Carson, D. F., and Shalla, L. The concept of demodulation in theorem proving. J. ACM 14,4 (October 1967), 698-709.

55 Zadeh, L. A. Fuzzy sets. Information and Control. 8 (1965), 338-353.

56 Robinson, G. and Wos, L. Paramodulation and theorem-proving in first-order theories with equality. In Meltzer, B. and Michie, D. (eds.). Machine Intelligence 4. Edinburgh University Press, Edinburgh, 1969, pp. 135-150.

57 Guard, J. R., Oglesby, F. C., Bennett, J. H., and Settle, L. G. Semi-automated mathematics. J. ACM 16,1 (January 1969), 49-62.

58 Huet, G. P. Constrained Resolution: A Complete Method for Type Theory. Ph.D. Dissertation, Case Western Reserve University, Cleveland, 1972.

59 Quinlan, J. R. and Hunt, E. B. A formal deductive problem-solving system. J. ACM 15,4 (October 1968), 625-646.

60 Sandewall, E. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. Advance Papers of Third International Joint Conference on Artificial Intelligence, Stanford, Calif., 1973, pp. 230-234.

61 Huet, G. Confluent reductions: abstract properties and applications to term rewriting systems. Technical Report, IRIA-Laboria, Rocquencourt, France, 1977.

62 Livesey, M. and Siekmann, J. Termination and decidability results for string-unification. Memo CSM-12, Essex University Computing Center, Colchester, Essex, Great Britain, August 1975.

63 Livesey, M. and Siekmann, J. T-unification. Technical Report, Universitat Karlsruhe Institut fur Informatik I, Karlsruhe, West Germany, 1976.

64 Lankford, D. S. and Ballantyne, A. M. Decision procedures for simple equational theories with a commutative axiom: complete sets of commutative reductions. Technical Report, Mathematics Department, University of Texas at Austin, March 1977.

65 Lankford, D. S. and Ballantyne, A. M. Decision procedures for simple equational theories with permutative axioms: complete sets of permutative reductions. Technical Report, Mathematics Department, University of Texas at Austin, April 1977.

66 Lankford, D. S. and Ballantyne, A. M. Decision procedures for simple equational theories with commutative-associative

axioms: complete sets of commutative-associative reductions. Technical Report, Mathematics Department, University of Texas at Austin, August 1977.