1

MICRO

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Tableau-Based Theorem Proving for Representation and Reasoning

by

Edward Shaw-Lee Suen, B.C.S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario

April 27, 1987.

© Copyright

1987, Edward Shaw-Lee Suen

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-39414-5

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

Tableau-Based Theorem Proving for Representation and Reasoning

submitted by Edward Shaw-Lee Suen, B.C.S.

in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science

Carleton University

Date

ABSTRACT

The representation and manipulation of knowledge is a crucial concern in artificial intelligence. Many formalisms have been advocated over the past two decades of research. Unfortunately, the informal approach of many of these proposals has detracted from their potential contribution.

Logic is a well-defined representation language that is a suitable candidate for the basis of new representation frameworks. In order to take advantage of the formal foundation supplied by logic, the lessons learned from the earlier proposals can be cast in a theorem proving framework.

Following this structured methodology, the system HARP (Heuristics Augmented Refutation Procedure) has been designed as a testbed for these ideas. To demonstrate the validity of this approach, a compiled version of HARP, optimized for efficiency, has been implemented and tested as a general purpose theorem prover. Favorable empirical results, such as the timely solution of Schubert's Steamroller, have been achieved and are described.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Professor Franz Oppacher, for his invaluable contribution to my intellectual and personal development.

My parents have been a constant source of encouragement and guidance. I am grateful for their concern and support.

TABLE OF CONTENTS

ACCEPTANCE SHEET.....	ii
ABSTRACT.....	iii
ACKNOWLEDGEMENTS.....	iv
TABLE OF CONTENTS.....	v
LIST OF FIGURES.....	viii
LIST OF APPENDICES.....	x
1 INTRODUCTION.....	1
2 LOGIC.....	8
2.1 Syntax.....	8
2.2 Semantics.....	11
3 AI AND LOGIC	16
3.1 Overview of Knowledge Representation.....	17
3.2 Classical Theorem Proving.....	20
3.3 The Logical Analysis of Knowledge Representation.....	26
3.3.1 Procedural/Declarative Controversy.....	28
3.3.2 Network-Based Formalisms.....	33
4 HARP - BACKGROUND.....	49
4.1 System Overview.....	49
4.2 Tableaux.....	56

5	HARP - SYSTEM DESCRIPTION.....	62
	5.1 The Basic Algorithm.....	62
	5.2 Proof Condensation.....	69
	5.3 HARP's Architecture and Heuristics.....	78
6	THE IMPLEMENTATION.....	95
	6.1 Implementation and Development Environment.....	95
	6.2 Program Details.....	98
	6.2.1 Categorizing the Formula.....	98
	6.2.2 The Indexing Scheme.....	99
	6.2.3 The Main Loop.....	103
	6.3 Empirical Results.....	106
7	MANY-SORTED LOGIC.....	111
	7.1 Overview of Many-Sorted Logic.....	111
	7.1.1 Syntax.....	115
	7.1.2 Semantics.....	117
	7.2 Many-Sorted Resolution.....	118
	7.3 Many-Sorted Tableaux.....	123
8	CONCLUSIONS.....	125
	8.1 Future Work.....	126
	8.2 Results and Observations.....	129

REFERENCES.....	133
-----------------	-----

APPENDICES.....	139
-----------------	-----

LIST OF FIGURES

Figure	Description	Page
1-1	The role of HARP within an intelligent agent.....	2
1-2	The architectural organization of HARP.....	6
2-1	Syntax and semantics.....	15
3-1	Semantic networks compared to logic.....	34
3-2	Frame-based classification refinement.....	38
4-1	HARP without admissible inference rules.....	61
5-1	Admissible inference rules.....	67
5-2	Triggering the admissible inference rules.....	68
5-3	Type A removal.....	71
5-4	Type B removal.....	72
5-5	Type A removal causing a Type B removal.....	73
5-6	Competing candidate closure nodes.....	75
5-7	Poor selection of closing nodes.....	77
5-8	Good selection of closing nodes.....	78
5-9	Embedded existential quantifiers.....	83
5-10	Selection between branching compounds.....	85
5-11	Minimization of additional parameters.....	86
5-12	Proof with admissible rules, example 1.....	87
5-13	Proof with admissible rules, example 2.....	87
5-14	Domain-specific rules.....	88
5-15	Nontheorem detection.....	90
5-16	Nontheorem detection with $\forall \exists$ patterns.....	92

5-17	HARP proof with heuristics, example 1.....	93
5-18	HARP proof with heuristics, example 2.....	94
6-1	Solution times for Schubert's Steamroller.....	109
7-1	Many-sorted axiomatization of Schubert's Steamroller.....	112
7-2	Sort hierarchy of Schubert's Steamroller.....	113
7-3	Walther's many-sorted calculus compared to FOL.....	119
7-4	Single or multiple most general unifiers.....	122
7-5	Nonempty sort condition.....	122

LIST OF APPENDICES

Appendix	Description
A-1	Bledsoe's problem
A-2	Monkey and banana problem
A-3	Schubert's Steamroller, version ***
A-4	Schubert's Steamroller, version **
A-5	Nontheorem detection

1 INTRODUCTION

This thesis describes the system HARP which is ultimately intended to be a cornerstone of the representation and reasoning components of an intelligent agent (see Figure 1-1). HARP employs logic to represent knowledge of problem solving situations. The manipulation of this knowledge is performed by a sophisticated architecture which combines a powerful, uniform proof procedure with specialized heuristics.

Logic is a language for encoding axioms describing a domain and the theorems resulting from these axioms. We will begin with a description of this language (Chapter 2) which emphasizes the formal definition of both the syntax and the semantics. Formal results describing the relationship between these two components of logic can then be developed from this stable foundation. The firm theoretical foundation of logic along with its metatheoretical results supports a clear understanding of its strengths and limitations as a knowledge representation language.

Chapter 3 contains an exposition of the role that logic plays within artificial intelligence (AI). A description of the classical theorem proving efforts dating from the 1960's is given. This research was aimed at the design of general purpose proof procedures which could verify the validity of arbitrary theorems expressed in first-order logic (FOL). Logic is employed as a representation language and the theorem provers were employed as the reasoning mechanisms. We will argue that logic plays a crucial role in the study of knowledge representation, both as a representation formalism itself and as a tool to analyze other representations.

In the early 1970's, researchers, dismayed by the poor empirical results of the classical theorem proving methods, investigated procedural representations and various indexing schemes to achieve increased efficiency (see section 3.3.1). A variety of representation schemes, such as semantic networks and frames (see section 3.3.2), were

concocted and advocated. The theoretical foundation of many of these schemes were questionable, and as a consequence, the description of these various methodologies were vague and inadequate.

Figure 1-1: *The role of HARP within an intelligent agent*

A great deal of confusion arose on questions comparing the technical merits of each approach. Only more recently, has it been realized that when viewed as representation formalisms, these various schemes can all be essentially captured within a logical framework. This and other recent developments mark the beginning of an important trend

to utilize logic not only as a tool to analyze representation formalisms but also as the basis for new formalisms (see for example the papers in the recent conference on *The Theoretical Aspects of Reasoning About Knowledge* [Halpern86]). By building on top of logic, it may be possible to utilize its sound theoretical foundation. Logic can be a primary building block of AI systems.

From the lessons learned from this earlier work we will derive adequacy criteria which can be employed to develop new knowledge representation frameworks. The important role which logic plays within AI indicates the need to investigate logic-based architectures. HARP is a manifestation of this requirement. Since it is ultimately intended to function in an AI environment, we have set forth the following design goals for this system: 1) an expressive representation language with well-defined semantics, 2) clear understanding of the inferential properties of this language 3) naturalness of the representation language and inference mechanisms, 4) indexing of pertinent knowledge, 5) efficiency, 6) ability to incorporate domain dependent heuristics for specialized behavior, and 7) detection of nontheorems.

The system HARP, presented in Chapters 4 - 6, is designed to satisfy these adequacy criteria. HARP retains the well-defined, declarative format and the favorable, formal properties of the classical theorem prover (eg. *soundness* and *completeness*, see section 2.2) while achieving a general purpose efficiency level that is empirically demonstrated to be at least that of its predecessors. The representation expressions (formulas) of HARP are structurally organized with an internal indexing scheme. This indexing scheme highlights the relevant expressions important for the effective operation of HARP. Efficient retrieval of pertinent expressions is facilitated. Heuristic control strategies which rely upon the information supplied by this indexing scheme produce a procedural type of advice indicating the most promising avenues of reasoning to follow.

These global control strategies have access to the entire problem solving situation so that proper deliberation on control can be conducted. The criticisms of logic which motivated the proceduralists to develop alternative representation formalisms are dealt with in this framework.

We will now present a quick overview and rationale of some of the criteria which we have employed as design goals for HARP.

Any powerful reasoning agent must have at its disposal a strong capability to represent the wide spectrum of knowledge that is necessary for this agent to effectively deal with the endless number of dissimilar situations it will encounter. FOL is a representation language which is widely acknowledged for its powerful expressive capability (see Chapter 3). HARP employs FOL as its representation language.

The system must operate in a very natural style. The internal representation and reasoning mechanisms must be elegantly simple so that it is easy for humans to follow the ongoing operations. This criterion is important for the development of interactive systems which may accept human input and behave as automated reasoning assistants to humans. External advice may occasionally be required to selectively guide the system through difficulties encountered during the execution cycle. A representation that is natural for humans eases the problem formulation task. Furthermore, an easily understandable reasoning mechanism supports the generation of clear explanations justifying this reasoning process. A discussion of the features of HARP which deal with this criterion is found in section 4.1. These issues are also encountered during the construction of applied reasoning systems such as MYCIN [Buchanan84].

Another more long term justification of the goal to achieve naturalness is the realization that eventually these intelligent agents will need to have some capability of performing a reflective, introspective style of reasoning (eg. [Doyle80, Smith82,

Davis82]). A declarative representation that is natural supports this kind of behavior. A clear reasoning mechanism will more easily yield useful observations when examined under this style of reasoning.

As previously mentioned, an indexing scheme structures the representation expressions in an organization that facilitates the retrieval of relevant pieces of knowledge. Expressions which are related to each other are explicitly linked together in this organizational structure. These links highlight the pieces of knowledge that are pertinent to the solution of the encoded problem. The retrieval of the relevant expressions are thus facilitated by this indexing scheme. In section 3.3, through the examination and comparison of previous representation proposals, the importance of indexing is demonstrated. The indexing scheme of HARP is described in section 5.1.

To achieve a high uniform performance level, powerful domain independent heuristic strategies have been developed to guide the deductive process (see section 5.3). In addition, HARP has a unique proof condensation or redundancy elimination procedure (see section 5.2) which vastly increases the performance of the proof procedure by compensating for possible shortcomings in these control strategies. The indexing scheme, the domain independent control heuristics, and the proof condensation procedure are key ingredients of HARP which supply the capability to function at a high level of performance on general purpose problems.

The architecture of HARP (see Figure 1-2) also allows the incorporation of domain specific heuristic advice to enhance the performance of specialized operations (see section 5.3). These domain dependent heuristics suggest the most promising avenues of reasoning to explore, and since they only reorder the application of inference rules, the property of *completeness* is neither affected by these heuristics nor by the general domain independent heuristics mentioned earlier.

Figure 1-2: The architectural organization of HARP

Since HARP is designed to function in an AI environment, it must be able to detect nontheorems. An experimental set of heuristics (see section 5.3) has been proposed for this task. Since logic is undecidable, a complete set of heuristics will never be finalized;

rather, the development of this set is an ongoing, incremental process. Heuristics of this type could quite conceivably affect the *completeness* property of HARP.

The development and expression of these three classes of heuristics (the domain independent control heuristics, the domain dependent control heuristics, and the nontheorem detection heuristics) is an ongoing process which requires constant modification to enhance performance, to specialize the behavior of the theorem prover, and to detect additional types of nontheorems. A meta-level architecture is proposed whereby these heuristics are explicitly represented (see section 5.3). This knowledge-based approach to theorem proving enables the application of proven expert system techniques, such as the iterative, incremental development and feedback methodology that has been successfully employed to produce systems which solve previously intractable problems (eg. see [Hayes-Roth83]). Experimental results supporting this architectural approach are presented in section 6.3.

Many-sorted logic and C.Walther's many-sorted resolution-based calculus are the topics of discussion in Chapter 7. An adaptation of the many-sorted unification algorithm is applied to enhance the system HARP for problems which have a natural many-sorted structure (section 7.3).

Finally, Chapter 8 will contain a summary of the primary contributions and observations of this thesis.

2 LOGIC

Logic arose from the efforts of philosophers to devise a formal calculus for stating and resolving arguments. Theories may be expressed and validated within this language. Many features of this formalism, such as the inference rule Modus Ponens (see section 2.1), model human-like reasoning patterns. The primary character of logic is its formally defined syntax and semantics. This solid theoretical foundation supports the derivation of various metatheories which describe these two components. In this chapter we will first present the syntax of classical first-order logic and then describe the associated semantics. There are several logic texts the reader may refer to which present a more detailed exposition of this subject matter (eg. [Jeffrey67, Smullyan68, Hunter71, Mendelson64]).

2.1 Syntax

The syntactic component of logic deals with symbols and the proper ordering of these symbols. This component is usually presented as the grammar of a formal language. It is a formal language because it can be defined without reference to the interpretation of any of its symbols. In other words, the language can be defined without ascribing meaning to the words of the language.

Any acceptable *word* is called a *well-formed formula*. The proper syntax of these well-formed formulas are described in terms of grammar rules. These grammar rules determine the universal set of all possible logic formulas. Another set of rules called the *inference rules* enable the specification of particular subsets of this universal set. These inference rules in combination with an initial set of formulas referred to as the *premises* are able to *derive* a subset language of this universal set. The derived words of these special languages are called the *theorems* with respect to the given premises.

The *alphabet* of the formal language is composed of the following symbols:

- 1) a set of constants, $C: c_1, c_2, c_3, \dots$;
- 2) a set of variable symbols, $V: x_1, x_2, x_3, \dots$;
- 3) a set of predicate symbols, $P: P_{n,1}, P_{n,2}, P_{n,3}, \dots, n \geq 0$;
- 4) connectives: $\Rightarrow, \wedge, \vee, \forall, \exists, \Leftrightarrow, \neg$
- 5) punctuation symbols such as $(,),$ and $","$

The set of terms T is defined to be the union of the set of variables V and the set of constants C . An *atomic formula* is a predicate symbol followed by well-defined terms as arguments. The following rules define a well-formed formula (wff):

- 1) an atomic formula : $P_{n,j}(t_1, \dots, t_n)$, where $n \geq 0, j \geq 1, P_{n,j}$ is any n th place predicate symbol, and $t_1, \dots, t_n$ are terms;
- 2) $(\neg A), (A \Rightarrow B), (A \wedge B), (A \vee B),$ and $(A \Leftrightarrow B)$, where A and B are also well-formed formulas; we often say these formulas are formed via *truth functional composition*
- 3) $\forall x(A), \exists x(A)$, where x is any variable and A is any well-formed formula

For convenience and clarity of presentation we will adopt some informal syntactic conventions. These conventions are fluid and are sometimes adjusted to fit the context of the presentation. Variable symbols will be represented by the symbols of the latter part of the English alphabet such as $u, v, w, x, y,$ and z . Constant symbols will usually be the initial symbols of the English alphabet such as $a, b, c, d, e, f, g,$ and h . We will sometimes also use lower-case strings to represent constants (eg. john). Predicate symbols will be denoted by capitalized letters or strings beginning with a capital. Because the presence of many parentheses and commas make the formulas hard to read, these punctuation symbols will often be omitted. The pair of parentheses surrounding the arguments of atomic formulas are usually left out and the commas separating these

arguments are also omitted. In cases of ambiguity, the following precedence binding hierarchy of the connectives will be employed to denote implicit parentheses. This hierarchy is listed from the strongest levels to the weakest.

$$\neg, \forall, \exists$$

$$\vee$$

$$\wedge$$

$$\Rightarrow, \Leftrightarrow$$

A *literal* is an atomic formula or a negated atomic formula. The scope of a quantifier is the succeeding formula delimited by the next set of matching parentheses. A *free* variable is a variable which is not in the scope of a quantifier containing that variable.

We will now define the language of *clauses*, a subset of the language of FOL which consists solely of formulas which are in *conjunctive normal form*. A formula in conjunctive normal form is a conjunction of conjuncts which themselves are disjunctions of literals. There are no existential quantifiers, and by convention, the universal quantifiers are not explicitly written; rather all free variables are deemed to be implicitly universally quantified. A process called *Skolemization* [Loveland78] translates existentially quantified variables to functional expressions. Thus, an additional set of functional symbols $F = \{f_{ij} \mid i, j \geq 1\}$ are needed for the language of clauses. A term for this language is now defined to be either a constant, variable, or a function. A function consists of a functional symbol f_{ij} followed by i terms. Again, for convenience and clarity of presentation, we will adopt the informal syntactic convention of using lower case letters such as f , g , and h to represent functional symbols.

As a digression, it should be noted that formulations of logic with functional symbols are equivalent to those without functional symbols. Any n -place function symbol can be duplicated with a corresponding $n+1$ -place predicate symbol.

The deductive apparatus of logic is usually based on a given finite set of inference rules. This set of inference rules may vary among different formalizations of logic. An inference rule supports the derivation of a new logical formula from combinations of existing formulas. The operation of any inference rule is effectively computable. Given any combination of formulas, F , there is a procedure which can effectively determine whether the set of formulas F satisfies the precondition of the given inference rule. If so desired, the inference rule may then be invoked to mechanically produce the derived result. Consider the well known inference rule *Modus Ponens*; let A and B be any well-formed formulas, then B is a consequence of the two formulas A and $(A \Rightarrow B)$. The notation $\phi \vdash B$ is used to indicate the syntactic derivability of the formula B from the set of formulas ϕ (possibly empty).

The syntactic and mechanical application of inference rules is a key feature and benefit of logic. The formulas of logic may be manipulated without regard to the underlying semantics. Deductive machinery can be constructed to perform these syntactic manipulations. The effective computability of these inference rules is crucial to the proper operation of this machinery.

These inference rules are said to be *sound* if the derived formulas are *valid* with respect to the underlying semantics (see section 2.2). In addition, these rules are said to be *complete* if every *valid* formula with respect to the semantics can be derived under these rules. It is a common practice when presenting a new deductive procedure to prove soundness and completeness statements about the inference rules comprising the procedure.

2.2 Semantics

Another aspect of first order logic is the formal definition of its semantics. The

semantics specify the intended meaning of the well-formed formulas defined by the syntax rules of the preceding section.

The formulas of logic are specified relative to a universe of discourse. Simply speaking, this universe is the world where the topic of discussion is situated. This universe consists of entities and the relationships between these entities. The entities and relationships of this world are a mirror reflection of the entities and relationships in the actual domain of discourse.

The semantics of logics are defined in a recursive fashion. The base foundation of this recursion is the *interpretation*. An interpretation consists of two parts: 1) a structure which is a description of the world and 2) an assignment which relates the symbols of the formal language to the entities and relationships in the description of the world.

The structure is a description of the world and specifies the entities of this world and the relationships among these entities. A structure ζ is composed of the following ingredients:

- 1) a non-empty set U called the universe
- 2) a distinguished set C of elements of U
- 3) a set of sets of n -tuples for $n \geq 0$

The second part of the interpretation is the assignment which relates the symbols in our formal language to the entities and relationships in the description of the world specified by a structure. Given a structure ζ and a formula A , an assignment ψ from A to ζ is described by the following mapping:

- 1) each free variable of A is mapped to an element of U
- 2) each constant is mapped to an element of the distinguished set C
- 3) each predicate symbol $P_{0,i}$, $i \geq 1$, is mapped to T or F
- 4) each predicate symbol $P_{n,i}$, $n, i \geq 1$ is mapped to a set of n -tuples

To illustrate, suppose we are describing a family which consists of four members: mom, dad, and two sons. There are four entities in this domain of discourse and there is a marriage relationship between the parents along with mom and dad relationships between the parents and the children. The following formulas outline the relationships mentioned in regard to this family, if the symbol 'a' corresponds to the father, 'b' denotes the mother, 'c' and 'd' represent the two sons, the predicate 'Married' signifies the marriage relationship, and similarly the predicates 'Mom' and 'Dad' represent their respective relationships.

Married(a,b)
Dad(a,c), Dad(a,d)
Mom(b,c), Mom(b,d)

An interpretation is essentially a mapping between the symbols (of the alphabet) in the formal language to the entities and relations in the description of the domain of discourse. A *valuation* function evaluates the formula under an interpretation, composed of a structure ζ and an assignment ψ , to yield the truth of the formula (i.e. T or F if the formula is true or false). The meaning of a formula is thus specified by its truth or falsity with respect to a particular world. The following rules specify this valuation function $\wp$ mapping from a well-formed formula which we will denote as Z to one of the truth values T or F:

let A and B be any wffs; let x be any variable;
suppose Z is of the form:

- 1) an atomic formula of the form $P_{0,i}$, $i \geq 1$: $\wp(P_{0,i}) = \psi(P_{0,i})$
- 2) an atomic formula of the form $P_{n,i}(a_1, a_2, \dots, a_n)$, $n \geq 1$, $i \geq 1$: if $(\psi(a_1), \psi(a_2), \psi(a_3), \dots, \psi(a_n)) \in \psi(P_{n,i})$ then $\wp(P_{n,i}(a_1, a_2, \dots, a_n)) = T$ else F

- 3) $(\neg A)$: if $\wp(A) = F$ then $\wp((\neg A)) = T$ else F
- 4) $(A \wedge B)$: if $\wp(A) = \wp(B) = T$ then $\wp((A \wedge B)) = T$ else F
- 5) $(A \vee B)$: if $\wp(A) = T$ or $\wp(B) = T$ then $\wp((A \vee B)) = T$ else F
- 6) $(A \Rightarrow B)$: $\wp((A \Rightarrow B)) = \wp((\neg A \vee B))$
- 7) $(A \Leftrightarrow B)$: if $\wp(A) = \wp(B)$ then $\wp((A \Leftrightarrow B)) = T$ else F
- 8) $\forall x (A)$: if for every assignment of entities in the domain to x , $\wp(A) = T$ then $\wp(\forall x (A)) = T$ else F
- 9) $\exists x (A)$: if for at least one assignment of entities in the domain to x , $\wp(A) = T$ then $\wp(\exists x (A)) = T$ else F

This version of logic is called first-order because it does not permit quantification over predicate or function symbols (i.e. unlike second-order logic). First-order logic allows only variables which range over the individuals of the domain to be quantified. Second-order logic generalizes these variables so that quantified variables may also range over the relationships (predicates) between the domain individuals. Third-order logic is a similar extension whereby the variables are now allowed to range over the predicates of second-order logic (i.e. the relationships between the relationships between the individuals of the domain).

A formula F is *satisfiable* if there exists an interpretation where F is true. Notice that F may contain *free* variables, i.e. variables which are not in the scope of a quantifier containing that variable. A *closed* formula does not have any free variables and is referred to as a *sentence*. From now on we will only be concerned with the subset of this formal language which consists only of the closed formulas (i.e. only the sentences).

A *model* of a formula F is an interpretation where F is true. Similarly, a model for a set of formulas S is an interpretation where every formula $F \in S$ is true under that interpretation. Intuitively speaking, a model is a description of the world corresponding to the set of formulas in S . The notation $\wp \models A$ is used to denote the semantic entailment of A from $\wp$. In other words, the formula A is true whenever the set of formulas

belonging to φ are simultaneously true.

A formula F is valid if it is true in every possible interpretation of F . Suppose A and B are well-formed formulas, $A \Rightarrow B$ is valid if B is true in every interpretation which A is also true. If a formula F is valid then as a consequence of the definition of our formal system, $\neg F$ is unsatisfiable (i.e. $\neg F$ is not satisfiable). The converse is also true; if F is unsatisfiable then $\neg F$ is valid. Proof procedures determine the validity, or analogously, the unsatisfiability, of formulas of this formal system.

For certain types of FOL proof procedures, provability is equivalent to validity. Semantic entailment, $\models$, is the same as syntactic derivability, $\vdash$. This is another way of saying that these proof procedures are *complete* and *sound*. They are complete in the sense that the proof procedure is capable of proving all valid formulas of this formal language. They are considered sound because every provable formula is a valid formula. Metatheoretical results, such as soundness and completeness, tie together the two main components of logic (see Figure 2-1).

Figure 2-1: Metatheoretical results such as soundness and completeness enable a more lucid understanding of the capabilities of the proof procedures of logic. These results tie together the syntactic component of logic (the symbols) to the semantic component (the meaning of these symbols)

3 AI AND LOGIC

AI attempts to extend the capabilities of the computer by pushing back the frontiers which limit the current applications. Through the incorporation of technology which enables the computer program to mimic some patterns of human reasoning, new applications previously considered to be unsolvable can be attempted. In this chapter we will outline the relationship between logic and AI. To properly describe this relationship, an overview of knowledge representation and the role it plays within AI is presented (section 3.1). *Knowledge representation* deals with the study of how to represent knowledge and reason effectively with it in the context of a computational environment.

It is argued that logic plays a crucial role in knowledge representation. First of all, logic, by itself, is certainly a viable representation language [Loveland78, Wos84]. In addition, it is often acknowledged within the AI research community that logic is the tool of choice to analyze the various approaches to representing knowledge [Newell82]. In fact, it is now impossible to work in the field of knowledge representation without some background in logic. For example, almost all of the authors in the proceedings of the recent 1986 conference on the *Theoretical Aspects of Reasoning About Knowledge* [Halpern86] employ logic in the presentation of their material.

The classical approach to theorem proving was motivated by the desire to utilize logic as a uniform, general purpose representation formalism. We will give a brief description of this *resolution-based* approach in section 3.2. A selected sample of the alternative representational approaches that have been proposed over the past two decades is then given in section 3.3. In presenting this material, we demonstrate how logic is crucial to the analysis and proper understanding of these various proposals. We follow the historical development of these approaches to elucidate the reasons for their invention. In particular, it is shown that the earlier rejection of logic in favor of more procedural

representations appears to be a premature gesture.

The arguments in this chapter dictate the importance of investigating logic. The solid theoretical foundation of logic supports the construction of additional mechanisms on top of this formalism to facilitate representation and reasoning. We are already witnessing manifestations of this approach in the expert systems arena where subsets of logic have been successfully applied as the basis for knowledge representation languages in constrained problem solving applications (eg. MYCIN [Buchanan84]). The system - HARP, whose ultimate goal is to operate as a problem solving component in a general purpose AI environment, is a product of this approach.

3.1 Overview of Knowledge Representation

A critical result of the last three decades of research is the realization of the importance of knowledge and thus the importance of how to represent knowledge. The solutions to the key problems of AI hinge upon the ability of the program to represent the right kinds of knowledge and the ability of the program to effectively manipulate this knowledge to generate new knowledge in the course of goal-directed problem solving behavior.

The study of knowledge dates back thousands of years to philosophers who have attempted to define and characterize this entity. Indeed the study of knowledge is a major subfield of philosophy called epistemology. Alan Newell discusses some of the issues in this area and how they are regarded by the AI community in his 1980 AAAI Presidential Address [Newell82]. *Knowledge* is defined in a behavioral sense. It is some entity that is instrumental in producing intelligent behavior which can be causally attributed to that entity. Knowledge representation languages are formalisms which attempt to explicitly capture this ethereal quantity. Brian Smith's *Knowledge Representation Hypothesis*

underscores this characterization:

"Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantic attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge." [Smith82, p.32]

In Newell's treatment, knowledge is represented at the symbol level by both symbolic expressions stored as the appropriate computer data structures and knowledge manipulation processes which perform interpretive and generative operations on these symbolic expressions (data structures). The data structures by themselves are inert. It is the amalgamation of both the active processes and the static data structures that produce meaningful activity. The procedures encoding the knowledge manipulation processes must interpret the data structures and transform them in a fashion which is consistent with their interpretation.

This delineation of knowledge into the two components of symbolic expressions and manipulative processes is necessary to deal with the problem that knowledge in an intelligent agent is potentially unlimited. For example, our knowledge of the world is exceptionally rich and varied so that we can accommodate a multitude of different situations. We adapt general principles to deal with specialized instances. We also generalize upon specific instances to hypothesize generalities. Our flexible and dynamic behavior is a direct result of these manipulative processes which enable us to react to an endless stream of new and changing situations.

Logic certainly appears to be a viable candidate to represent knowledge. The amalgamation of the representation formulas along with a given deductive apparatus has

the potential to produce intelligent behavior. One of the primary motivations which fueled the strong interest in the classical theorem proving research of the 1960's and early 70's was the goal of employing logic (and more specifically first-order logic) as a knowledge representation language.

For example logic has been applied to a wide spectrum of problem solving tasks which include solving puzzles, logic circuit design, logic circuit validation, research in mathematics, real-time systems control, and program debugging and verification. The reader interested in sample descriptions of logic-based systems operating on these specific problems may consult [Wos84]. Much of the work on automatic programming has been centered around the use of logic (eg. [Boyer79, Bibel82, Biermann84]). Logic has been employed to implement deductive retrievers acting as question answering systems [Green69]. The list of applications is long and serves to demonstrate the utility and versatility of this formalism as a representation language.

As previously mentioned, alternative representation methodologies were proposed because of the inefficiency of logic when applied in its pure primitive form. The two primary metrics employed for comparisons between these competing methodologies center around the *expressive adequacy* and the *heuristic power* of the representation formalism.

Expressive adequacy measures the potential ability of the symbols in the formalism to represent various pieces of knowledge. The ease of expressing this knowledge is also an important criterion of this measure. Many authors incorrectly overlook this concern. This is unfortunate as we will now explain.

Treating deduction as computation, somewhat like in PROLOG, it can be shown that the Horn Clause subset of FOL is Turing machine equivalent [Kowalski79]. This means that any computation that can be performed on a Turing machine can also be

formulated in terms of Horn clauses and computed with the sl-resolution proof procedure (see section 3.2 and also [Loveland78]) which is both sound and complete for this subset of FOL. Consequently, FOL is also Turing machine equivalent. Without considering ease of expressiveness, most of the representation languages (such as frames, FOL, Horn clauses, second-order logic, modal logic, and LISP) would all have the same expressive power - which is obviously counterintuitive.

The representation should enable efficient access (retrieval) which should highlight the important pieces of knowledge that are potentially relevant to the current reasoning activity. The efficiency and effectiveness of the manipulation operations are directly addressed when evaluating the heuristic power of these methodologies.

In passing, we should notice that these two measures are not orthogonal. The ease of expressiveness is related to the conceptual level of the representation. To illustrate, the closer the representation is to the entities and relationships of the domain, the easier it is to capture this domain information. In turn, the conceptual level will impact the ease with which the representation expressions are to be manipulated. The conceptually higher level expressions will support manipulation operations which directly reflect the operations of the domain while those of a lower level would be further removed and obscured from the actual domain. Extra operations could possibly be needed to reach the same effects. To summarize, a higher level representation is easier and more natural to manipulate than a correspondingly lower one. The manipulation operations of the higher level representation are also likely to be more powerful.

3.2 Classical Theorem Proving

Note that the defined semantics of logic do not provide a constructive proof theory (this problem actually gets worse for other augmented logics such as modal logic where

the search for inference rules is still an active research concern). Indeed first-order logic is undecidable. There does not exist an effective decision procedure to determine the validity of arbitrary formulas. However inference rules can be specified to form sound and complete proof procedures for first-order logic. The proof procedures utilizing these methods are partially decidable as they will eventually return with a proof if the input formula is valid. Unfortunately, it is not theoretically possible to place bounds on the time this process will take.

For over two decades, most of the effort in the theorem proving community has been placed on developing proof procedures based on resolution [Robinson65]. An intense period of activity followed the introduction of the original procedure. The results of this activity affected all areas of computer science, and in particular AI.

During the 1960's and for a large part of the 1970's, theorem proving was synonymous with AI. As mentioned before many influential people in the AI community believe that logic is the universal language which ties together and unifies the concepts in AI [Nilsson80, McCarthy69, Hayes77, Moore82, Charniak85, Kowalski79]. Because of the expressiveness of logic, many forms of automated reasoning can be described within the theorem proving paradigm. For example, planning can be expressed as a theorem proving exercise in which case the answer will come back in the variable instantiations. MYCIN, the prototypical expert system example, can also be seen as a special example of theorem proving technology. Even programming can be viewed as theorem proving. Indeed the notion of computation is closely allied to deduction. Prolog, a logic programming language that is the basis of the Japanese 5th generation computing efforts, is simply a resolution theorem prover operating on Horn clauses (a subset of the language of clauses).

As alluded to above, it is also believed that parallel and distributed hardware

implementations of this theorem proving technology will form the basis of the AI machines of the future. To realize this vision distributed, models of computation are currently being investigated. For an example of the issues and concerns in this area of work, see [Santoro86].

It should be clear to the reader that the work performed in the theorem proving mainstream resulted in many tangible benefits. We will give a brief overview of some of the work performed on this classical approach to theorem proving. For a more detailed exposition the reader is advised to see [Loveland78].

Resolution is a rule of inference which operates on a syntactically restricted language of first-order logic called the language of clauses (see section 2.1). The formulas of this language must be in conjunctive normal form. A formula is a conjunction of a set of conjuncts called clauses. Each clause is a disjunction of a set of literals (an atomic formula or a negated atomic formula). There are preprocessing procedures to convert from the language of first-order logic to the language of clauses.

In the ground resolution logic (i.e. the propositional or variable free case), the clauses are compared for complementary literals (i.e. literals differing only by a $\neg$ sign); and if found, a new clause is generated by combining all the literals of both clauses minus the two complementary literals. For example, in the following figure, the third clause is the result of resolving the first two clauses. The A_i 's and B_i 's denote atomic formulas without any arguments.

$$\begin{array}{l}
 (A_1 \vee \neg A_2 \vee A_3 \vee \dots \neg A_n) \\
 \wedge (\neg A_1 \vee B_1 \vee \neg B_2 \vee \dots B_m) \\
 \hline
 (\neg A_2 \vee A_3 \vee \dots \neg A_n \vee B_1 \vee \neg B_2 \vee \dots B_m)
 \end{array}$$

The basic resolution proof procedure is a refutation procedure whose goal is to

demonstrate the unsatisfiability of the given set of input clauses. Thus the user of such a system would negate the proposed theorem before feeding it into the system. The proof procedure operates in an iterative fashion searching for direct evidence of the hypothesized inconsistency of the input. The following is an outline of the basic proof procedure:

Basic-Resolution-Loop

Repeat

- 1 find two resolvable clauses -- clauses which contain a corresponding pair of complementary literals
- 2 apply the resolution inference rule to the two selected clauses
- 3 add the new clause to the existing set of clauses

Until empty clause is derived

End Basic-Resolution-Loop

The procedure systematically applies this resolution inference rule to the expanding set of clauses until the unsatisfiable empty clause (a clause which does not contain any literals) is derived.

The *unification* algorithm is used to lift the proof procedure to the general case. The unification algorithm is intuitively a clever universal instantiation procedure embodying the principle of least commitment. It can also be viewed as a bi-directional matching procedure where patterns are allowed on both of the data to be matched. Given two literals the algorithm computes (if possible) a special set of substitutions so that both literals will be identical. This special set of substitutions are a most general unifier in the sense that any other set of unifying substitutions can be duplicated by composing this most general unifier with some set of substitutions (i.e. the least commitment principle). The purpose of this algorithm is to determine the most general set of substitutions which

would produce a complementary pair of literals and therefore a pair of resolvable clauses. Since the substitutions produced are the most general unifiers, redundant substitution and resolution operations are minimized.

A substitution σ is a function which maps terms to terms and can be described by a set of expressions of the form:

$$\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, \dots, x_n \leftarrow t_n\}$$

These expressions, when applied to a formula, are interpreted as t_i replaces all occurrences of x_i . Let $\text{DOM}(\sigma) = \{x_i \mid x_i \neq t_i\}$ and $\text{COD}(\sigma) = \sigma(\text{DOM}(\sigma))$. ϵ will denote the identity substitution. A substitution σ is a *unifier* of a nonempty set of literals D if $|\sigma D| = 1$. A most general unifier of D is a substitution σ which satisfies the following criteria:

- 1) σ is a unifier of D
- 2) if ϑ is a unifier of D then there exists a substitution λ such that $\vartheta = \lambda \circ \sigma$

It has been proved (most notably by J.A. Robinson [Robinson65]) that if a unifier exists for a set of literals D then there exists a most general unifier. The following unification procedure from [Rich83] will calculate a most general unifier if it exists.

Unify (L1, L2)

if L1 or L2 is a symbol
then

if L1 and L2 are identical then return NIL endif

if L1 is a variable

then

if L1 occurs in L2 then return unsuccessful unification endif

return $\{L1 \leftarrow L2\}$

else if L2 is a variable

```

    then
        if L2 occurs in L1 then return unsuccessful unification endif
        * return {L2 ← L1}
    else return unsuccessful unification endif
endif

if length(L1) is not equal length(L2)
then return unsuccessful unification endif
set SUBST to NIL      {SUBST will contain the substitutions used to unify
                       L1 and L2}

for i = 1 to number of elements in L1 do
    call unify with the ith element of L1 and the ith element of L2, putting
    the result in S
    if S = unsuccessful unification then
        return unsuccessful unification endif
    if S is not equal to NIL then
        apply S to the remainder of both L1 and L2
        SUBST := append(S, SUBST)
    endif
return SUBST
End Unify

```

Resolution has been attractive to many people because of its simplicity. Since there is only one inference rule, the control problem (i.e. the problem of deciding what to do next) associated with natural deduction is avoided. As a consequence many variations and refinements have been developed over the past twenty years (see [Loveland78] for a list and descriptions). These additions are mainly concerned with the selection of the most suitable pair of clauses to resolve upon. The development of effective search strategies is crucial to the success of these systems.

3.3 The Logical Analysis of Knowledge Representation

In the AI research community there exist many proposed solutions to the representation problem of determining the most suitable data structure to encode the knowledge of a system. The representation should enable efficient access (retrieval) and should also highlight the important pieces of knowledge that are potentially useful in the course of reasoning activity. Other criteria such as the expressive adequacy and the ease of expressiveness are also dependent upon the proper choice of data structures. Each of the various approaches has their own individual combination of strong and weak points. The selection of certain data structures will enable particular inferences to be accomplished very efficiently while others may not be feasible.

It is currently a subject of debate whether any of these candidate approaches are clearly superior in an overall context. However it is possible to characterize each of these formalisms against the backdrop of logic. The most suitable situations for the different representations are more clearly depicted by characterizations which bring out the strengths and weaknesses of each approach. To illustrate these notions more clearly, we will describe some of the major approaches and characterize them against logic in order to elucidate a common ground for comparison. Logic is the benchmark which is used to compare the different representation formalisms.

First-order logic is generally accepted as a very expressive knowledge representation language. Expressiveness deals with the ability to represent knowledge. It refers to the range of concepts that can be represented and the ease with which these concepts can be captured within the representation expressions of the language. For example, FOL is more expressive than the majority of the associative network formalisms which really have only the expressive power of propositional logic (the variable-free subset of FOL). The reader is advised to refer to section 3.3.2 for a look at the translation

of semantic networks into their equivalent formulas of logic. This expressive capability and the formal definition of its semantics are key factors in the selection of logic as the knowledge representation benchmark.

The backlash against logic was due in part to the empirical inefficiency of the early resolution-based theorem provers which attempted to employ logic in its pure unaugmented form as the framework for a general purpose problem solving mechanism. As we will argue, this criticism was unfounded because in actual fact, the logical foundation was not at fault but instead the naive application and implementation of this tool was to blame.

Another area of criticism about logic has been directed at the idealized worst case inefficiency of its deductive mechanisms resulting from the expressiveness of its knowledge representation capability [Levesque84a]. However many people feel the ideal knowledge representation language needs at least the expressiveness of first-order logic. Indeed, several researchers are investigating augmentations of classical logic (eg. temporal logics [McDermott82, Allen84]; modal logics for representation of belief [Konolige84]; non-monotonic "logics" [McCarthy80, McDermott80, Reiter80]). Thus, on the surface, there appear to be conflicting forces within the knowledge representation community.

Fortunately, this conflicting situation may be resolved by realizing that the worst case metric is often not appropriate within the realm of AI systems. For example, we ourselves as humans often have a great deal of difficulty performing simple tasks. Yet, we can solve other more difficult tasks quite easily. Our *overall* behavior is quite productive and efficient. One of the primary goals of HARP is to improve the overall (average case) efficiency of theorem proving by utilizing specialized heuristic strategies for the most frequently performed inference patterns. These heuristic strategies will

embody both general deductive advice and domain specific knowledge. The most frequent deductions are carried out efficiently with the aid of these strategies.

We hope this section will impart to the reader a sense that much of the current criticisms of logic are unfounded and collapse under careful scrutinization. By supplying a discussion detailing the logical analysis of knowledge representation, the reader will hopefully gain an understanding of the utility of logic and appreciate the stance taken by the many prominent researchers who believe that logic is the universal language of AI. Only a logical characterization of the many competing representation languages serves to supply some order in the chaotic arena of knowledge representation research.

The other knowledge representation formalisms are poorly understood because of their vague, ill-defined semantics. No counterparts to the metatheoretical results of logic such as soundness or completeness exist in these alternative approaches.

These critical factors dictate the need to employ logic as the basis for a knowledge representation methodology. The ideal approach would build on top of logic to utilize its firm formal foundation supplied by the rigorous definition of both its syntax and semantics. This approach should be able to exploit the centuries of research which have yielded the important metatheoretical results describing logic. The system HARP is designed to meet these criteria.

By following the historical development of a selected sample of representation formalisms, the lessons learned from this earlier research will be summarized. These lessons point the way towards design goals for a new representation framework. These design goals have directed the development of HARP.

3.3.1 Procedural/Declarative Controversy

Logic is the prototypical example of a declarative representation. The

representation expressions (i.e. the logic formulas) are explicitly declared in a high-level uniform manner. A set of general inference procedures are typically associated. Knowledge expressed in this fashion can be utilized in a multitude of ways. To illustrate, the knowledge employed by the system MYCIN is expressed in a declarative horn clause like subset of logic. This knowledge is used to perform diagnosis, to provide explanations, and to aid in knowledge acquisition [Buchanan84]. Declarative representations are usually described as modular and flexible. Declarative knowledge is like a *knowing that* [Winograd75]. It is more of a static description that is given independent of how the knowledge is supposed to be used.

Procedural representations were developed because of the dismay over the empirical inefficiency of the early resolution-based theorem provers (this inefficiency was incorrectly presumed to be an intrinsic characteristic of logic). Procedural knowledge emphasizes the aspect of *knowing how* to use the knowledge [Winograd75]. It is characterized by procedures (computer programs), which have interactions between each other. The expressions of this representation (i.e. the procedures) are able to facilitate specialized operations very efficiently. These operations are performed efficiently because of the explicit interactions between the expressions (i.e. control flow and common data areas). There is, however, an associated loss of modularity in this trade-off.

A direct line of reasoning is performed whereby irrelevant knowledge and false avenues of reasoning are avoided. However, the expressions of the procedural representations are less transparent than the declarative representations. The expressions are intertwined and thus less modular. There is no formal account of the semantics of the language expressions, unlike the declarative representations typified by FOL. It is the combination of both the representation expressions and the manipulation mechanisms

(procedural execution) which yield indications of the underlying knowledge. The procedures have no meaning when examined independently of their manipulation mechanisms. The underlying knowledge is obscured within the code and is implicitly utilized by the reasoning process.

To illustrate the preceding statements, a discussion of the alphabetization example [Barr81] will now be presented. When a computer program sorts words based on their alphabetical order, it utilizes knowledge of the form that *A* precedes *B*. This knowledge is usually implicitly encoded as a test between the numerical codes corresponding to the letters. On the other hand, knowledge about *how* to alphabetize is explicitly represented. In a declarative system, the opposite situation exists. Knowledge about the alphabetical order is explicitly represented but knowledge of how to perform the alphabetization is implicit within the inference rules of the representation.

Dismayed by the failure of the general theorem proving techniques to function in situations more complex than the typical toy scenarios, Carl Hewitt and his colleagues at MIT developed the language PLANNER for procedural representation [Hewitt71]. PLANNER was a goal driven, backtracking logic-based language quite similar to PROLOG (incidentally the MIT group claim PROLOG was a direct derivation of their efforts [Hewitt85]) that was developed as an attempt to incorporate procedural advice to control deduction. This representation language is a product of an effort to build upon logic. Unfortunately, the architecture of the language suffered from some design deficiencies that led to its eventual demise. These deficiencies will be outlined in conjunction with the succeeding description of the system.

PLANNER had primitives to specify control information. The sequential order of the specified language expressions affected the computational properties of the system. This programming language-like mechanism can be contrasted to logic where the order of

the formulas have no effect. In PLANNER, the programmer would order his logic expressions so that a more efficient solution would result from the expedition of pertinent inferences.

Information attached to the implication connective, $\Rightarrow$, could indicate whether the expression was to be used in a forward or backward manner. In addition, there was the capability to recommend groups of expressions which were relevant to proving certain goals. These devices, along with the ordering of the expressions, added procedural semantics to the language. Direct lines of reasoning could often be supported through the utilization of the procedural semantics. In these cases knowledge relevant to the problem solving situation is quickly retrieved and pertinent inferences are performed.

The primary drawback of PLANNER is that only local methods of control are employed. Control decisions are made only with respect to localized information sources. For example, specific theorems are designated for forward chaining. The priority of inferences are determined by the order of the formulas within a local context (similar to the clause ordering mechanism of PROLOG). Instead, global control strategies based on the overall problem solving situation are needed to facilitate the proper selection of actions. The assessment of new situations should be performed on a dynamic basis which incorporates all possible information sources. The control strategies of HARP have at their disposal, the entire problem solving situation. This application of global control strategies differentiates HARP from PLANNER.

Even though PLANNER is based upon logic, the incorporated extensions confused and obscured the semantics of the language. For example, THNOT, a connective of the language, is somehow involved with non-monotonic reasoning and the closed world assumption [Reinfrank85], but no formal, well-defined semantics have yet been presented which can satisfactorily explain the full meaning of this connective.

The inference mechanism of PLANNER is not *complete* (in a loose sense) with respect to its declarative semantics. There are cases where the system would have enough information to accomplish its goals but would be unable to because of its restricted inference mechanisms. The designers of PLANNER accepted this trade-off in an attempt to achieve greater efficiency. However, it is our belief that this trade-off is not necessary, and instead, both goals of efficiency and completeness can be achieved. HARP, for example, is able to apply heuristic advice within a declarative and complete logical framework.

This procedural/declarative controversy later led to the development of the frame notion [Minsky75, Winograd75, Schank77]. By attempting to combine the best of both worlds, this new representation formalism was seen by its proponents as an endeavor to resolve this controversy. Procedural representation expressions were integrated within declarative structures via a technique called *procedural attachment*. The resulting synthesis quickly gained popularity among the AI research community.

Frame systems are part of a class of knowledge representation and manipulation mechanisms based upon a network-like formalism. These systems all have an indexing scheme in common which appropriately structures their forms so that special inferences are accomplished efficiently. The structure of the indexing scheme is often explicitly displayed along with the expressions of the representation in a graphical link and node notation. The following section is based on a discussion of these systems. The characterization of these systems with logic will offer us a more lucid understanding of their strengths and limitations, which in turn will allow us to formulate detailed criteria and design goals for new representation frameworks.

3.3.2 Network-Based Formalisms

We will use the term *associative networks* to denote representations such as semantic networks and frames where the links of the indexing scheme are modelled after particular relationships. The objects of the domain are treated as the nodes. Thus the expressions of these representations do not have a flat organization but are instead well-connected so that the access of relevant information may be facilitated by the explicit links.

The indexing scheme of these associative networks is not based solely on syntactic information unlike that of connection graph schemes employed by some theorem provers [Kowalski75, Stickel82]. Instead, the expressions of these network-based formalisms are structured in a manner which reflects associations employed by the knowledge manipulation mechanisms. For example, the expressions of frame systems are typically structured around the objects of the domain. This object-oriented structuring is used explicitly by the knowledge manipulation mechanism to facilitate efficient specialized inferences such as *inheritance* (described below). The links of the indexing scheme can be interpreted and generated by the knowledge manipulation processes and are not, for example, generated because of the presence of complementary literals. These links are present because of the underlying semantics of the representation expressions. We will sketch a tentative proposal for incorporating a semantic-style indexing scheme for the logical expressions of HARP in section 4.1.

Semantic networks were originally devised by Quillian [Quillian67] as an attempt to encode word definitions from a dictionary into a computer representation. This representation was primarily designed as a psychological model of human language behavior and consisted of a complex network of words (nodes) and interconnecting relationships (links).

A number of interesting results were achieved in this seminal work. Quillian developed an inferential process, called spreading activation search, which relied heavily on the indexing scheme of links. Spreading activation search enabled the system to contrast and compare different words (nodes). Essentially, the network is traversed from the two nodes to be compared, and the intersecting path is taken to be the answer.

The problem with this approach is the ill-defined semantics of the representation. Quillian defined the meaning of a node to be "all the nodes that can be reached by an exhaustive tracing process, together with the total sum of relationships (links) among these nodes" [Quillian 67]. William Woods strongly criticized the vague semantics of this representation in his influential paper "What's in a Link" [Woods75]. He makes the point that meaningful inferences are hard to develop and propagate with this definition. Indeed, it was almost as if the reasoning program was still operating on the dictionary. Computer programs could not reason with the information in a dictionary, nor could they really do many constructive things with Quillian's representation.

It is only a more recent discovery that much of the intended meaning of the semantic network formalisms can be captured within logic. The following figure shows the semantic network equivalent of the fact that John's grade in cs100 was 85, which is expressed in logic as `Grade(john,cs100,85)`.

Figure 3-1: The semantic network equivalent of `Grade(john,cs100,85)`. (adapted from [Levesque84a])

Since the semantic network formalism only contains analogues of unary and binary predicates, the existence of objects such as the grade assignment *g-a1* would have to be postulated in order to carry out a proper translation. This translation appears to have obscured and complicated the original expression. To illustrate, the direct logical translation of the semantic network is the following significantly longer expression :

$$\text{Grade-assignment}(g-a1) \wedge \text{Student}(g-a1, \text{john}) \wedge \\ \text{Course}(g-a1, \text{cs100}) \wedge \text{Mark}(g-a1, 85)$$

The basic semantic network formalism is equivalent to only the propositional subset of FOL. However, there exist augmentations of the basic formalism so that variables and quantifiers may be employed [Schubert76, Hendrix79]. Nevertheless, the loss of conciseness and naturalness demonstrated by the above example is magnified even more by these extended semantic network formalisms.

Much of human reasoning relies on the guidance of stored knowledge. We very rarely compute new knowledge structures from scratch. Instead we rely on our vast library of previous experiences to deal with new ones. Often, it appears that reasoning involves the determination of the most suitable previous experience and using our memory of that experience to function in the current situation. Our memory of the old experience enables us to place expectations on the current situation. If these expectations do not bear out, then the expectation violation may remind us of another similar experience which can be recalled and utilized in the same manner.

The frames theory was devised to capture this style of reasoning. A frame is the structure capturing knowledge of the previous experience. The frame corresponding closest to the current situation is determined (often called *partial matching*) and then used to interpret this ongoing setting. Unexpected events are handled by following links to

other frames. Reasoning is thus a process which proceeds from frame to frame following links when new situations or demands arise.

An influential paper [Minsky75] written by Marvin Minsky introduced this idea of frames and frame systems which later formed the basis for a new generation of knowledge representation systems (eg. KRL [Bobrow77], UNITS [Stefik79], KL-ONE [Brachman85b]). In this paper, Minsky sketches an impressionistic vision based upon prototypes, defaults, multiple perspectives, analogies, partial matching (also called classification), and inheritance. As previously alluded, part of the appeal of this new approach is the closer correspondence to cognitive memory models than that achieved by previous representations (eg. mathematical logic and semantic networks).

Minsky describes a frame to be a

"... data structure for representing a stereotyped situation... Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed. We can think of a frame as a network of nodes and relations. The top levels of a frame are fixed and represent things that are always true about the supposed situation. The lower levels have many terminals - slots that must be filled by specific instances or data. Each terminal can specify conditions its assignments must meet."
[Minsky75]

Frames are knowledge representation structures which are especially suited to object-oriented description. The term object-oriented is generalized to include abstract entities such as stereotyped situations and concepts as well as physical objects. A frame has associated with it a group of important elements called the slots. Slots may be treated as relationships which involve that particular object or frame.

A frame can be viewed as a fancy record structure where each record slot is associated with possibly several aspects which describe the slot [Charniak80]. These descriptions may include among other things, the slot value, slot type, and slot demons (procedures that are automatically executed when triggering conditions are satisfied). Frames are often described in terms of this slot and filler notation which is especially conducive to the description of objects where the slots denote attributes of the object and the fillers denote the attribute values. The slots are usually filled by other frames which can describe other objects.

Procedures are often attached to slots, called *procedural attachment*, so that a repertory of operations can be accomplished efficiently. For example, default values may be associated with slots so that in the absence of specific information, these values may be utilized as defaults. These default values are calculated through procedures attached to the slots, referred to as if-needed demons [Winston81, Charniak80]. Procedures can also be executed when slots are filled with values, if-added procedures. To illustrate, the commercial knowledge system building tools such as ART and KEE employ this device under the name of *active values*. A popular use of this feature is the direct manipulation of screen icons. If we have a frame describing a temperature gauge, an if-added procedure attached to the temperature slot could manipulate an icon depicting a temperature gauge. When the temperature changes, the temperature gauge icon reflects these changes immediately. One of the most appealing points of the frames approach is this ability to intermix procedures along with the more declarative knowledge.

Inheritance, a common feature of frame-based systems, is a good example of a tailored inference mechanism specially designed to take advantage of the indexing scheme. Special slots which are often marked by the *is-a* label denote a class/subclass relationship. If we can imagine the lattice of frames partially ordered by this link to be a

tree-like network starting from the topmost more general classes downwards to the more specialized subclasses, then inheritance would be an inferential operation which would bring about the downward propagation of properties and values from the generalized classes to the specialized ones. Exceptions can be specified to override inheritance from the more general classes, if it is so desired.

Figure 3-2: Classification by successive refinement (adapted from [Fikes85])

Frame-based classification refinement is a technique that has been successfully applied in the expert systems arena. Diagnosis can be performed via this technique. To illustrate, symptom attributes of the diseases are arranged in a hierarchical fashion starting from general disease classes to specific disease instances. Inheritance is propagated downwards through this hierarchy so that more specific disease classes will have the option to carry the attributes of its superior superclasses. The partial matching procedure tries to find the frame in this hierarchy which most closely correlates to the conditions of

the particular problem situation. Starting from the most general disease class, the procedure works its way down the hierarchy until the most specific frame is found (see Figure 3-2 for a pictorial description).

Frame systems correspond closely to semantic networks and in fact can be viewed in some sense as a semantic network as observed in [Rich83]. However, frames generally have a more complicated internal structure and a larger variety of manipulation mechanisms (eg. procedural attachment, partial matching). They have an associated style of processing (reasoning) that is usually not attributed to semantic networks.

A common criticism of network-based formalisms is their inadequate description which leads to a poor understanding of the underlying semantics [Brachman83, Woods75]. In fact, the following editorial comment is found in the knowledge representation anthology [Brachman85a] as a prologue to Minsky's well-known but very informal article [Minsky75] which introduced the frame notion:

"This problem of vagueness and general lack of rigor has unfortunately followed many of those who pursued the frame ideas, as if the topic itself demanded a certain style of research."
[Brachman85a, p. 245]

Because of the vagueness and impressionistic style of the literature on frames, there are a number of possible interpretations of this theory [Hayes79]. One interpretation is that frames are a representation formalism like logic, which can be used to express knowledge. In this context, frames should be compared to the other alternative representation formalisms in terms of the range and ease of expressiveness with which knowledge can be captured. In addition, the manipulation mechanisms associated with the formalism should be judged by their capability to potentially infer new knowledge. These criteria determine the *expressive adequacy* of the formalism which is the prominent

concern of this interpretation.

Another interpretation of the frames theory is that it is a statement of how to structurally organize the representation expressions so that retrieval and inference operations can be efficiently realized. In this heuristic interpretation [McCarthy69], the manipulation mechanisms rely upon the indexing scheme of the representation language to realize the goals of the reasoning program in a timely fashion. Not only is it important that inferences can potentially be accomplished, but also that their production be direct and quick.

This interpretation can be viewed more as an implementation concern which is to be dealt with after the issue of expressive adequacy is addressed. For example, the organizational structure offered by various indexing schemes can be built on top of logical formalisms. Logical formalisms can be implemented as lists; they can also be implemented as networks with links between formulas or subformulas. These approaches display different computational properties but they all rely on the same representation medium. However, this is not to say ~~that~~ the indexing scheme is unimportant. In fact, we believe that the indexing structure is crucial to the effective performance of any powerful reasoning program.

Yet another interpretation of frames is the metaphysical interpretation [McCarthy69]; stating that frames make a certain assumption about the entities of the world. The use of frames implies that a certain kind of knowledge is to be represented by them. This knowledge, often referred to as prototypical knowledge, is employed to represent commonly occurring stereotypical situations such as *dining out in a restaurant* or *shopping in a supermarket* [Schank77]. A certain style of reasoning is typically associated with this knowledge and it does appear that this kind of knowledge is frequently present and utilized. The metaphysical interpretation of the frames theory

supports the hypothesis that this kind of knowledge and style of reasoning is necessary for an effective AI program. However, this interpretation is separate from such issues as how this library of prototypical knowledge can actually be represented and effectively employed.

In this discussion we shall only be concerned with the first interpretation that frames are a representation formalism. Keeping this interpretation in mind, it may be prudent to digress a bit before outlining the logical translation of the frames theory. As previously discussed, FOL is Turing machine equivalent. It is a straightforward observation to make that the representation and manipulation mechanisms of frames can be formulated in terms of logic. The main contribution of [Hayes79] is the demonstration that most of the frames theory can be directly translated into logic with little trouble. We will now present a discussion which will hopefully demonstrate this simple and natural translation.

Pat Hayes in [Hayes79] outlines the formalization of the frame ideas in terms of logic-based concepts (also found in [Nilsson80, Charniak85]). This formalization clears up many of the vagueness and possible misconceptions of the frame concept. In his paper, it is argued that without a formal account it is not really possible to have a concrete understanding of this technique. One of the key roles of logic is to serve as a tool for the analysis of the various competing knowledge representation schemes.

In this particular analysis provided by Hayes, a frame corresponds to the *entity* found in logic. Each slot of the frame is analogous to a binary relation whose first argument is the entity corresponding to the frame containing the slot and whose second argument is the value of the slot. For example, consider the following frame describing the class of houses:

House

Is-a	Building
Kitchen:	-----
Bathroom	-----
Living-room	-----

The straightforward logical translation of this frame is as follows:

$$\begin{aligned}\forall x(\text{House}(x) \Leftrightarrow & (\text{Building}(x) \\ & \wedge \exists y \text{ Kitchen}(x,y) \\ & \wedge \exists y \text{ Bathroom}(x,y) \\ & \wedge \exists y \text{ Living-room}(x,y)))\end{aligned}$$

The symbols *House*, *Building*, *Kitchen*, *Bathroom*, and *Living-room* are predicates. This logical formula describes all individuals of the domain belonging to the class of houses. Notice that the *Is-a* slot denotes a class/subclass relationship.

Most of the specialized inference mechanisms found in frame systems are easily duplicated within logic. One often found inference pattern in frame systems is called the *criticality* condition. This inference pattern is used to determine whether a given entity is an instance of the class description. Referring to the translation of the house frame, the logical equivalence connective enables us to deduce that this entity is an instance of the class of houses if it is a building which contains a kitchen, bathroom, and living room.

Inheritance is another inference pattern that is typically attributed to frame systems. This inference strategy is an automatic benefit of a logic-based description of frames. Using the same house example, we can see that the normal class/subclass inheritance is easily captured by logic. Since all houses are buildings, therefore any relationships involving buildings are also applicable to houses.

One advantage that frame systems have over a flat list of logic formulas is the increased efficiency with which the frame system is able to realize this inheritance

inference. Notice, however, that much of this advantage disappears with the system HARP which relies upon its indexing scheme and specialized inference strategies to capture aspects of this capability. The reader is advised to see sections 5.1 and 5.3 for a description of these features.

A more complicated inference strategy is the frame-based classification refinement technique. Here a number of the other inference patterns such as the criteriality condition are utilized within a partial matching routine. Much of this behavior can also be easily captured within logic. However, because of the vague literature descriptions, it is not entirely clear what functions the actual frame-based mechanisms perform. This is of course another benefit of formalization within logic, which requires complete clarity and distillation of the concepts and notions of the theory. Essentially, in the logical translation, the criteriality condition is iteratively employed to determine the most specific frame which corresponds to the current situation.

However, frame systems have an implicit reflexive description capability which is perhaps beyond the ability of first-order logic. For example most frame systems have a default capability which allows them to make default attributions *unless* some condition follows from the current state of knowledge of the world [Hayes79, Reinfrank85]. The *unless* connective is stated within the context of the current state of knowledge. An examination of the knowledge encoded in the internal representation expressions is necessary to carry out this kind of inference. The system would have to be aware of its own state of knowledge to determine the applicability of these default inference rules.

Defaults provide trouble for classical first-order logic. This non-monotonic style of reasoning is clearly beyond the scope of the classical logics and requires the development of new logical formalisms to capture the nuances of this reasoning pattern (see [Reinfrank85] for a survey of these issues). However, the recent development of

non-monotonic reasoning formalisms casts this kind of reasoning in a clearer light (eg. see the papers from the *1984 Workshop on Non-Monotonic Reasoning and Artificial Intelligence* and also from Vol. 13, No.2, *Artificial Intelligence* - special issue on non-monotonic reasoning). It should be noted that a reasonable computational implementation of these proposals requires at least a partial ability of the inference engine to detect consistency or satisfiability. Some mechanism to detect nontheorems is crucial to this style of reasoning and is one of the long term goals of the system HARP.

Much of commonsense reasoning involves some sort of reflexive reasoning. The reasoner deals with personal limitations when conducting everyday affairs in the world. Awareness of resource limitations is important to effective human performance. Frame systems appear to have elements of this style of reasoning; however in this unformalized state its full potential cannot be realized until a more lucid understanding of this mechanism is achieved. These current attempts to formalize non-monotonic reasoning are a first step toward this very important goal. An ability to detect nontheorems plays a crucial role in non-monotonic reasoning. We feel that much more work in the automated theorem proving community should be directed towards the detection of nontheorems for this reason and others mentioned in section 4.1.

We have attempted to provide a brief description of the vision of frames and a broad outline of the logical translation of most of the ideas within this vision. Many authors in the AI research community support the application of logic as the universal basis for knowledge representation [Charniak85, Nilsson80, Hayes74, Hayes79, Moore82, McCarthy69]. Evidence of the applicability and universality of logic also manifests itself in the commercial systems which are targetted for the applied AI marketplace.

For example, ART is a commercially available expert systems building tool which

has a frame facility that is part of its knowledge representation language. The implementation of this frame facility exploits the relationship between frames and logic. In fact, ART compiles the user-specified external frame syntax into an internal logic-based representation. This commercially available system is a practical example based on the above ideas.

We have extolled the virtues of the network-based representations and emphasized their close correspondence to cognitive models. However, an important disadvantage of these formalisms is their inability to capture much of the incomplete knowledge representation capabilities of logic. Logic is necessary to express incomplete knowledge. Consider the following facts (adapted from [Levesque84a]) which can be expressed in FOL:

$$1) \neg \text{Student}(\text{john}).$$

John is not a student but we still don't know his occupation.

$$2) \text{Parent}(\text{sue}, \text{bill}) \vee \text{Parent}(\text{sue}, \text{george}).$$

One of Bill or George is Sue's parent, but we don't know which one.

$$3) \exists x \text{Cousin}(\text{bill}, x) \wedge \text{Male}(x)$$

Bill has a male cousin but we don't know who it is.

$$4) \forall x \text{Friend}(\text{george}, x) \Rightarrow \exists y \text{Child}(x, y)$$

All of George's friends have a child but we don't know George's friends nor do we know their children. George may not even have any friends and therefore it is possible that there are no children.

The network-based formalisms described in this chapter have difficulty expressing the kind of incomplete information that FOL is so well suited to represent. FOL can be

used to capture the important information of the domain without having to represent the details. It is this ability to specify incomplete pieces of knowledge that supplies the extra expressive power of FOL. This expressive power of FOL makes it an attractive basis for new representation frameworks.

We will now summarize the highlights of this chapter in order to distill the important contributions which are to be gleaned from this previous research work.

The earlier theorem proving research effort faded when the combinatorially explosive search spaces prohibited the solution of more substantial real-world problems. A school of thought primarily situated at MIT strongly criticized this declarative approach using general reasoning methods and subsequently a more procedural approach to knowledge representation was advocated. This procedural approach emphasized *knowing how* to perform specific functions efficiently while the declarative approach emphasized *knowing that* which enabled a general repertory of operations to be accomplished. Thus the MIT approach concentrated on specific procedures which could be efficiently applied to realize the desired behavior. The subsequent development of the PLANNER language reflected these attitudes.

Because of this emphasis on specialized inference methods and the poor results of the earlier theorem proving research, the use of logic decreased and the ideas associated with logic fell into disfavor. It is only more recently that people have begun to realize that logic, itself, was not to blame for these earlier research problems [Hayes77]. The primary contribution of logic is to ascribe a meaningful and well-defined formal semantics to the language expressions. Indeed, because the network-based formalisms suffer from a lack of well-defined semantics, considerable confusion obfuscates the exact nature and contribution of these representation proposals. Well-defined semantics are crucial to the development of mature and effective representation theories [Hayes74].

Logic does not prohibit further augmentations which enable specialized deductions. Thus the valid criticisms of the proceduralists were misconstrued as an attack on logic [Hayes77]. It makes no sense to throw away logic and start all over again, instead it makes more sense to build new schemes on top of logic. Indeed, this is a trend that is now manifested in the recent literature toward this use of logic (eg. [Levesque84b, Patel-Schneider86]). Furthermore it is our position that indexing schemes which are of a non-syntactic nature, such as those incorporated by frame systems, can be built upon logic to implement reasoning mechanisms which explicitly rely on the extra-logical features of this indexing scheme to facilitate efficient, specialized inferences. The realization of both the importance of logic as a tool to analyze and understand the plethora of representation approaches and as the basis for a representation, itself, foreshadows an important emerging trend.

We will now attempt to summarize the lessons learned from this earlier period of research:

- 1) Well-defined semantics are necessary to support a proper understanding of the representation formalism.
- 2) The representation language must be expressive.
- 3) Both the representation language and inference mechanisms must be natural.
- 4) FOL is the preferred representation to express incomplete knowledge.
- 5) Indexing schemes are necessary -- both syntactic and semantic.
- 6) Heuristics are necessary to encode a procedural-type advice to enhance the efficiency of specialized operations.
- 7) The ability to detect nontheorems is crucial.

The theorem proving architecture described in this thesis has been developed to meet the design goals which are derived from these lessons learned from this earlier research period of AI. We will now present a description of the system HARP.

4 HARP - BACKGROUND

This chapter describes a complete theorem prover for first-order predicate logic that combines a modification of the semantic tableau technique from [Smullyan68] with declaratively expressed heuristic strategies and with a redundancy elimination or proof condensation procedure. The resulting proof procedure HARP (Heuristics Augmented Refutation Procedure) is "natural" and easy to use for people as well as being computationally efficient. HARP's proof procedure is "natural" in the sense that it accepts the entire, unmodified language of predicate logic, avoids conversion to any kind of canonical form, uses quantifier instantiation instead of unification, and constructs its proofs according to natural-deduction-like inference patterns (see section 4.1). As will be pointed out in sections 4.1 and 6.3, it is also efficient: HARP solves the original first-order version of Schubert's Steamroller Problem in a time frame comparable to solutions using a many-sorted version of the problem and, indeed, seems to be the only theorem prover that can solve the unmodified, nonclausal version of this problem at all ([Cohn85], [Walther85]).

Section 4.1 discusses the design goals that have influenced HARP's construction and explains why we have chosen the tableau method as the (subsequently modified and enriched) foundation for HARP's proof procedure. Section 4.2 outlines the method of analytic tableaux and states the basic inference rules.

4.1 System Overview

Since HARP is intended to be used both interactively and as an inference engine for Artificial Intelligence applications, its construction is influenced by the overall design goals of achieving 1) utilization of an expressive representation language with well-defined semantics, 2) clear understanding of the inferential properties of this

language, 3) naturalness, 4) efficiency, 5) indexing to eliminate irrelevant premises and to highlight pertinent ones, 6) partial specifiability of the control structure by heuristic rules, and 7) rapid detection of certain types of nontheorems.

1-2) Logic is an expressive representation language with formally defined semantics which support a fairly clear understanding of its inferential properties. A logical formalism would serve to meet goals 1) and 2). The arguments dictated in Chapter 3 strongly suggest a logic-based representation language. Other competing formalisms are not as expressive and have vaguely defined syntax and semantics.

3) One of our main goals is to construct a human-oriented theorem prover, i.e. one that primarily addresses the needs and preferences of people rather than the presumed efficiency demands of machines. This goal of naturalness dictates the use of the full, unmodified language of first order logic and a natural style of proof construction.

To avoid having to make and support detailed psychological claims as to which notations and inference patterns people find "natural", we characterize the notion of naturalness in terms of a very crude but serviceable criterion: a notation or an inference style is natural if it is widely used as a didactic medium in introductory logic texts or if it is commonly used in the proofs and metaproofs of logicians. Although quite crude, this criterion suffices to rule out any notations based on canonical forms such as clausal form and any inference styles based on resolution, connection graphs or matrix matings.

It seems to us that, according to our criterion, a human-oriented theorem prover should use the full language of first order logic and should implement either a system of natural deduction (like the one in [Kalish64]) or a tableau system. We have opted for tableaux because we feel that, apart from their other advantages (see below), they admit of

a simpler control structure: unlike natural deduction systems they do not need to maintain any machinery for formally distinguishing between assumptions and assertions and for introducing and discharging assumptions.

Our algorithm seems to be unique in that it requires no conversion to any canonical form and accepts its (fully parenthesized) input without any preprocessing. Bibel remarks in [Bibel83] that the avoidance of clausal form constitutes a major advance in automated theorem proving but his system [Bibel82, Bibel83] still presupposes that negations are driven inward until they apply only to atoms. Even nonclausal resolution [Murray82] which is claimed to be least restricted in truth-functional form, requires quantifier-free formulas.

The two major disadvantages of the transformation to clausal form required by many other theorem provers have been pointed out by various authors (see e.g. [Andrews81],[Nilsson80], [Moore82]). First, by splitting a given premise into several, apparently unrelated clauses, conversion to clausal form not only diminishes readability but may also lead to a loss of potentially valuable control information and make it difficult to attach heuristic advice to individual premises. Second, the use of clausal form - in particular, the combination of prenexity and distributivity - leads to multiplicative growth of the number of literals. For example, converting the formula $(p \wedge q \wedge r) \vee (s \wedge t \wedge u)$ to clausal form yields 18 literals (9 clauses with 2 disjuncts each). HARP, on the other hand, will never generate more than 6 nodes for the literals present in the original formula (and since it decomposes nothing before embarking on a proof it might not even need to decompose the formula at all).

A natural style of proof should deploy principles of inference commonly used by people. Therefore we have opted for normal quantifier instantiation rules instead of Skolemization and unification and for natural-deduction-like rules instead of resolution or

connection graph methods which are favored in current approaches (e.g. [Andrews81], [Bibel82], [Loveland78], [Murray82], [Stickel82]).

We feel that the method of analytic tableaux, especially in the form given to it by Smullyan in [Smullyan68], provides an extremely natural and elegant codification of logic. Many introductory logic texts (eg. [Jeffrey67]) rely on tableaux as a didactic medium. Even Robinson, the inventor of resolution and the concomitant clausal forms, uses tableaux to introduce the idea of resolution [Robinson81]. The tableau method also allows for a very lucid semantic adequacy proof ([Fitting83]).

The natural flavor of the tableau calculus is due to the fact that its rules are syntactic reformulations of semantic evaluation rules. In addition, the tableau calculus is a paradigm of a cut-free or analytic calculus, i.e. to prove a sentence p one only considers its weak subsentences where a weak subsentence of p is either a subsentence of p or the negation of such a subsentence. This contrast with resolution systems, in which the length of a derived clause may exceed that of either parent clause, is exploited to provide increased efficiency in our system. The analytic property also makes it possible to keep the basic control structure of the algorithm simple. Furthermore, the tableau calculus can be easily extended to modal and other nonclassical logics (see e.g. [Fitting83]).

4) The addition of an indexing scheme and heuristics to the basic control structure of analytic tableaux and the avoidance of clausal form result in a highly efficient proof procedure.

HARP has done all the examples in [Smullyan68], several examples from [Bledsoe83] and a varied selection of roughly 100 theorems from chapters 1 - 4 of [Kalish64]. The average time for these proofs is less than one second. HARP's empirical efficiency is further demonstrated by the solution of traditional test problems for theorem

provers such as the monkey and banana problem [Nilsson80], the plane geometry problem [Loveland78] and, in particular, Schubert's Steamroller [Walther85].

Schubert's Steamroller is a challenge problem that had been unsolved for 6 years until recently in [Walther85]. The quickest solution time of 7.11 seconds [Walther85] was achieved by using a many-sorted clausal version of the problem (a further but apparently unimplemented refinement of this approach, a polymorphic many-sorted logic, is described in [Cohn85]). Solutions not using a many-sorted version of the problem have also been reported [Walther85]. Without using special knowledge, Stickel's theorem prover has a reported solution time of 2 hours and 53 minutes [Cohn85]. The ITP system has a reported solution time of 660 seconds while using a limited many-sorted type logic [Walther85], [Cohn85].

HARP is able to solve the original, nonclausal version of Schubert's Steamroller in 14 seconds on a Symbolics 3600. The resulting proof tree has 48 closed branches (see also section 6.3 and the appendix) and is built without any special purpose reasoning mechanisms and without any domain-dependent knowledge. It should be noted that the techniques used in converting traditional resolution theorem provers to many-sorted logic are equally applicable to tableaux. For example, the instantiation mechanism could be restricted by using a sort hierarchy of the universe (see section 7.3 for details). It is interesting to note that HARP has a solution time comparable to theorem provers that do use these techniques. Furthermore, HARP seems to be the only theorem prover able to solve the original, first-order version of the problem.

5) Since HARP is intended to work in an AI environment, it has to cope with situations in which there are many irrelevant premises and frequent requests to prove nontheorems. An indexing scheme allows HARP to select relevant premises and

heuristics are used to recover quickly from many attempts to prove nontheorems.

The indexing scheme helps in selecting relevant premises by making both syntactic and semantic information available to the theorem prover. The indexing scheme uses syntactic information such as the number of branching connectives in a formula and the presence of quantifiers and suitable candidate terms for instantiation to deactivate those premises that could not possibly contribute to a proof while highlighting those premises that could. This use of the indexing scheme as a syntactic filter is described in detail in section 5.1.

A few remarks must suffice to sketch some semantic uses of the indexing scheme. The basic idea is to constrain proof construction with semantic and world knowledge. Our current approach works roughly as follows: each predicate in a desired conclusion may be associated by hand with a range of predicates that are semantically related to it as its synonyms, near synonyms, components in its definition, etc. From a set of premises syntactically relevant to a given conclusion those premises are selected for initial consideration that also contain predicates which are semantically related to predicates in the conclusion. The semantic and world knowledge associated with predicates occasionally also serves to fill information gaps by supplying trivial and thus unstated premises (this is described further in section 5.3).

6) Because of the experimental nature of the use of heuristics in theorem proving, a control structure modifiable by a set of explicitly represented heuristics is preferable to a rigid architecture where the heuristics are hardwired into the control structure. (It should be noted that the timing results reported in b) have been achieved by a version of HARP with "hardwired" heuristics described in Chapter 6.) The heuristics can be classified into the following three groups (see section 5.3): heuristics to enable an efficient and

humanlike proof construction, heuristics to detect nontheorems, and domain-dependent heuristics (e.g. heuristics for transitivity, equality, commutativity, etc.). We also feel that this modifiable control structure is crucial to an investigation of strategy - learning theorem provers (eg. [Cohen81]). (However this issue will not be discussed in this thesis.)

7) To date, little work has been done by the automated reasoning community on the use of heuristics to detect nontheorems (for a rare exception, see [Schonfeld83]). We believe that the ability to quickly detect certain types of nontheorems is at least as critical to the success of a theorem prover in an AI environment as is its ability to weed out irrelevant premises. Section 5.3 contains a brief discussion of this issue and presents some heuristics that try to decide nontheoremhood on the basis of global properties of branches of proof trees.

These design goals are realized in a novel architecture for theorem provers that consists of a base-level and a meta-level.

The base-level executes a modified version of Smullyan's algorithm and performs bookkeeping tasks, i.e. it maintains and updates the indexing scheme, closes trivially inconsistent branches and applies basic inference rules to decompose the formulas indicated by the meta-level.

The meta-level first matches the heuristic rules against the nodes and the indexing scheme produced by the base-level. This results in a conflict set of all applicable heuristics. Then selection criteria (see section 5.3) are matched against the conflict set of heuristics, and a suitable heuristic is chosen. The effect of executing the selected heuristic is to notify the base-level as to which node is to be decomposed next.

This cycle - base-level produces nodes $\rightarrow$ meta-level heuristically selects a node to be decomposed $\rightarrow$ base-level produces more nodes - continues until either all branches of the proof tree are closed off or until some branch is declared, often under heuristic guidance, to remain open (see section 4.2).

4.2 Tableaux

HARP is based neither on variants of the resolution principle nor on natural deduction but on tableau methods: it implements a version of the semantic idea that proving a formula amounts to an unsuccessful attempt to construct a falsifying model for it. Tableau methods emphasize a systematic search for semantic counterexamples: a failed attempt to refute $\neg\Phi$ generates enough information to put together a countermodel to Φ .

The method of analytic tableaux provides a comprehensive test for consistency and, hence, for inconsistency, logical truth, and logical entailment. To prove that Φ is logically true, $\neg\Phi$ is assumed to be semantically consistent or satisfiable. Similarly, to prove that $\Phi_1, \dots, \Phi_n$ imply Φ , the set $\{\Phi_1, \dots, \Phi_n, \neg\Phi\}$ is assumed to be satisfiable. From the satisfiability assumption smaller and smaller (weak) subformulas are derived. If the assumption of semantic consistency is substantiated, a counterexample to the claim that Φ is logically true or that $\Phi_1, \dots, \Phi_n$ imply Φ can be directly retrieved from the constructed model. However, if the assumption of consistency turns out to be untenable, then the method will ascertain this in a finite number of steps. In other words, the method is complete.

It is convenient to represent models by binary trees. The set of sentences $\{\Phi_1, \dots, \Phi_n, \neg\Phi\}$ to be tested for consistency is put at the root of the tree. The set is consistent if there is at least one truth-value assignment to all the atomic subformulas of $\Phi_1, \dots, \Phi_n, \neg\Phi$ on which the latter are simultaneously true. The consequences of the

assumption that $\{\Phi_1, \dots, \Phi_n, \neg \Phi\}$ is consistent are then developed systematically, in a manner that ensures that no opportunity for generating contradictions is overlooked.

The tree is grown by entering on its branches the results of applying heuristically selected decomposition rules (see below) to sentences previously entered. This process continues until atomic subformulas of $\Phi_1, \dots, \Phi_n, \neg \Phi$ or negations of such are reached. When a root node is labelled with a consistent set, the repeated application of decomposition rules generates a tree with at least one consistent branch, i.e. a branch that does not contain both some atomic sentence and its negation. Conversely, if every branch of the tree is inconsistent, the original set is inconsistent as well.

The decomposition rules are similar to natural deduction elimination rules. In HARP, the similarity to natural deduction is further strengthened by our inclusion of "admissible rules", i.e. inference patterns such as Modus Ponens or Modus Tollens (see sections 5.1 and 5.3). When a decomposition rule is applied to any compound sentence that is not a universal quantification the compound is checked off to prevent further rule applications to it, and its subformula or subformulas are attached to the end of each open, i.e. consistent branch which passes through the compound. As soon as a contradiction turns up on a branch, the latter is said to be closed and marked with a "*". A pair of conjugate literals responsible for the contradiction is called a closure point.

A tree is said to be closed if all its branches are closed, and is said to be open otherwise. A branch is said to be complete if it contains either no unchecked compounds or the universal quantifications on it have already been instantiated with respect to all individual parameters appearing anywhere on the branch. A tree is said to be complete if all its branches are complete.

Any complete, open branch provides an interpretation under which the sentences at the root are simultaneously satisfied. Such an interpretation may be useful for certain

question-answering tasks. Different complete, open branches may provide different interpretations. The interpretation determined by a branch results from assigning the truth-value **T** to the unnegated atomic sentences on the branch and **F** to the negated ones.

HARP also enters dependency information into the trees it builds in order to record for each node all the nodes on which that node depends. Dependency information can be used in applications that involve nonmonotonic forms of reasoning (eg. see [Reinfrank85]).

In the tree calculus all sentences are classified as either elementary, i.e. atoms or negations of atoms, or as of type α , β , γ or δ . Sentences of type α , the conjunctive type, are those truth-functionally complex sentences from whose truth one can uniquely infer the truth values of their immediate subsentences. (For example, $\neg(A \vee B)$ is of type α because from its truth follows the truth of both $\neg A = \alpha_1$ and $\neg B = \alpha_2$). Sentences of type β , the disjunctive type, are all the other truth-functionally complex sentences. (For example, $A \Rightarrow B$ is of type β because its truth implies the truth of either $\neg A = \beta_1$ or $B = \beta_2$.) Sentences of type γ , the universal type, are those quantificationally complex sentences from whose truth one can uniquely infer the truth values of their instantiations. (For example, $\forall x Fx$ is of type γ because it implies all of its instances such as $Fa = \gamma[\tau]$.) Sentences of type δ , the existential type, are all the other quantificationally complex sentences. (For example, $\exists x Fx$ is of type δ which does not imply its instances such as $Fa = \delta[\tau]$.)

The semantic rationale for this classification lies in the fact - used in the completeness and correctness proofs - that for each interpretation a the following holds:

- a satisfies $\alpha \Leftrightarrow a$ satisfies α_1 and a satisfies α_2 ;
- a satisfies $\beta \Leftrightarrow a$ satisfies β_1 or a satisfies β_2 ;
- a satisfies $\gamma \Leftrightarrow a$ satisfies $\gamma[\tau]$ for each parameter τ ;
- a satisfies $\delta \Leftrightarrow a$ satisfies $\delta[\tau]$ for some parameter τ .

It is perhaps the most remarkable property of the tableau approach that its basic syntactic decomposition rules precisely mirror the downward ($\Rightarrow$) direction of the above semantic evaluation rules. The natural style of proof construction in HARP is largely due to this property.

In the following, a "|" in a rule indicates a branch in a tree. To avoid the need for relettering bound variables our parameters are syntactically distinct from the variables of quantification. The rules are to be understood as follows: from a formula of the structure shown above the line, derive a formula (or formulas) of the structure shown below the line.

Type α :

$$\begin{array}{cccc}
 \frac{}{\top} \Phi & \Phi \wedge \Psi & \frac{}{\neg(\Phi \vee \Psi)} & \frac{}{\neg(\Phi \Rightarrow \Psi)} \\
 \Phi & \Phi & \neg\Phi & \Phi \\
 & \Psi & \neg\Psi & \neg\Psi
 \end{array}$$

Type β :

$$\begin{array}{ccccc}
 \frac{}{\Phi \vee \Psi} & \frac{}{\Phi \Rightarrow \Psi} & \frac{}{\neg(\Phi \wedge \Psi)} & \frac{}{(\Phi \Leftrightarrow \Psi)} & \frac{}{\neg(\Phi \Leftrightarrow \Psi)} \\
 \Phi \mid \Psi & \neg\Phi \mid \Psi & \neg\Phi \mid \neg\Psi & \Phi \mid \neg\Phi & \Phi \mid \neg\Phi \\
 & & & \Psi \mid \neg\Psi & \neg\Psi \mid \Psi
 \end{array}$$

Type γ :

$$\begin{array}{cc}
 \frac{}{\forall\lambda\phi(\lambda)} & \frac{}{\neg\exists\lambda\phi(\lambda)} \\
 \phi(\tau) & \neg\phi(\tau) \quad \text{with proviso (1).}
 \end{array}$$

Type δ :

$$\begin{array}{cc}
 \frac{}{\exists\lambda\phi(\lambda)} & \frac{}{\neg\forall\lambda\phi(\lambda)}
 \end{array}$$

$\phi(\tau)$ $\neg\phi(\tau)$

with proviso (2).

Proviso (1): Let $\phi(\zeta)$ be any schema and ζ any variable which may or may not occur free in $\phi(\zeta)$. Let $\phi(\tau)$ be the result of replacing all free occurrences, if any, of ζ in $\phi(\zeta)$ by the term τ . The rule of universal instantiation (UI) permits the inference from $\forall\zeta\phi(\zeta)$ to $\phi(\tau)$, provided that no free occurrence of ζ in $\phi(\zeta)$ is within the scope of an occurrence of $\forall\tau$ or $\exists\tau$. This restriction prevents illegal capturing of the instantiating variable by a quantifier. (For example, this restriction blocks the invalid inference from $\forall y(Hyz \Rightarrow \exists z(Gz \wedge Jyy))$ to $(Hzz \Rightarrow \exists z(Gz \wedge Jzz))$.)

Proviso (2): Let $\phi(\zeta)$ and $\phi(\tau)$ be as before. When a quantifier is dropped by existential instantiation (EI) the instantiating term τ must be an individual parameter new to the entire branch. In the implementation, it is easier to use a constant foreign to the entire tree. (This "new parameter" condition prevents, for example, the invalid inference of $\exists x(Fx \wedge Gx)$ from $\exists xFx$ and $\exists xGx$.)

In principle, a universal quantification is instantiated with respect to every individual term anywhere on its branch unless such an instance is already on the branch. In practice, HARP relies on its full indexing scheme and heuristics (see sections 5.1 and 5.3) to try first only those instantiations which are most likely to contribute to the early closing of a branch. Only if no individual parameter has appeared is a new one chosen for the instantiation. Unlike all other compounds, a universal quantification is not checked off and, thus, can be used repeatedly.

The example in Figure 4-1, which is in fact a HARP proof without "admissible rules" (see sections 5.1 and 5.3), illustrates both the tableau method and the necessity to constrain the basic algorithm with heuristics.

Figure 4-1

In this example, any step other than EI in line 6) and the choice of any other instantiating terms for any of the UI applications would have led to wasted effort. An obvious heuristic sees to it that existential instantiations are usually carried out before universal ones and another heuristic picks instantiations which are likely to lead to early closure. A more subtle heuristic that tries to get at embedded existential quantifiers is responsible for saving some effort by using UI, in line 7), on 2) rather than 1). (Our heuristics are presented in section 5.3).

5 HARP - SYSTEM DESCRIPTION

Section 5.1 describes our modification of the tableau method and the indexing scheme that organizes the database of formulas and supports the application of heuristics. A detailed description of the proof condensation procedure and some examples of its use are presented in Section 5.2. Section 5.3 sketches HARP's architecture and presents a representative sample of our heuristics, together with several example proofs.

5.1 The Basic Algorithm

The tree calculus described above has guided the design of the basic loop of HARP:

Main-Loop

Initialization

Loop until proved or nontheorem

 Select a node from the priority queue

 Decompose the node -- prioritize the descendants using heuristics

 if a branch closure results

then condense the proof

 if there are more branches

then prepare for the next branch

else proved

endif

endif

 if current branch is complete, according to heuristics

then nontheorem

endif

Endloop

End Main-Loop

In the preparatory stage each formula is internally represented as a node. All formulas are put into a completely parenthesized format by making the connective precedences explicit. This enables efficient, $O(1)$ time, node decomposition. All decomposable nodes are kept on a priority queue. The priority of a node is determined via heuristics (see section 5.3 for examples). These particular heuristics use the information held by the indexing scheme.

The indexing scheme is implemented on top of a connection-graph-like mechanism [Kowalski75]. Links are kept between node pairs which consist of corresponding conjugate literals. Conjugate literal links are determined using a matching procedure which is a relaxed form of unification. Universally quantified variables match anything; constants match only the same constants; existentially quantified variables match only further occurrences of themselves within the same scope. Any literals that are unifiable will also match using our procedure. However since we do not eliminate quantifiers and in particular do not skolemize the formulas, links will occasionally be made between nonunifiable literals (eg. $\forall y \exists x Fyx$ and $\forall z Fzz$ will match but not unify). The difference between our matching procedure and unification is due to the presence of existential quantifiers.* It should be remembered that achievement of our design goal of naturalness forces us to avoid unification. The following procedure match describes the algorithm:

Match (atom1, atom2)

 if the predicate symbols of atom1 and atom2 are not the same
 then return no match endif

 loop for each argument arg1 in the argument list of atom1
 and for each argument arg2 in the argument list of atom2

* if neither of arg1 and arg2 are universally quantified variables

```
and they are not the same existentially quantified  
variables within the same scope  
and they are not identical to each other  
then return no match endif
```

```
endloop
```

```
return successful match
```

End Match

It should be noted that the entire match process is implemented efficiently because each variable is flagged to indicate its type when the formula is entered as input. Universally quantified variables are flagged with a 'U' to facilitate rapid identification. Existentially quantified variables are flagged with a pointer to the enclosing existential type formula which quantifies that variable. The actual checks are performed using the LISP 'eq' function.

Each link of the indexing scheme is labelled with a weight referred to as the link strength. The link strength is currently calculated by a heuristic using the formula $\lceil 32/2^n \rceil$ where n is the combined sum of the β split level of each literal. The β split level of a subformula is the number of β node decompositions necessary to reach the subformula from the top level formula. For example, in the formula $(P \wedge Q \Rightarrow R) \wedge G$, the β split level of both P and Q is 2, while that of R and G is 1 and 0, respectively.

Since in a dyadic tree branching increases exponentially with the height of the tree, we have decided to express the strength of a link as inversely exponentially related to the number of β node decompositions necessary to reach the closure point specified by that link.

If one of the pair is a γ node then the link will supply a candidate constant for universal instantiation when our matching procedure matches the universally quantified

variable with the constant. In contrast, [Wrightson84] colors the link with the most general unifier. The same heuristic is used to calculate the link strength. For example, given nodes $\forall xFx$, $\neg Fb$, Fa , the γ node $\forall xFx$ forms only one link to $\neg Fb$. b is also specified as a candidate constant along with the link strength 32. No link to Fa is built because instantiation with a would not lead to closure.

The current selection criterion which determines the priority of a node simply sums up the link weights of all the links associated with the node.

Decomposition of a node is done with the rules in section 4.2, optionally together with admissible inference rules (see below). When a universal formula is decomposed, our universal instantiation procedure is called to produce a constant which is literally substituted (i.e. using a modified version of the Lisp Subst function) for occurrences of the universally quantified variable.

It should be noted that since the formulas are completely parenthesized, there is no unnecessary memory allocation or formula duplication when α or β nodes are decomposed. Instead the new nodes point toward their respective subformulas within the ancestor formula which is itself left untouched. This sharing of structure among many nodes differs markedly from the production of new clauses in resolution-based theorem provers.

β node decompositions spawn a new branch of the tree. The ability to return to the state of computation at which the branch was created is provided by remembering the changes made since the most recent branch point. This is done by modifying all the procedures which change the global environment to record their change if the previous value was installed prior to the last branch point. Thus each object in the environment is actually a pair consisting of the value of the object along with its branch point indicator. Upon return to this branch point, the changes made since that branch point are undone.

Thus instead of saving the entire proof state, only the changes are noted.

Whenever a ground literal is produced via decomposition, a check for a branch closure is made. This is also done efficiently with the information contained in the indexing scheme: the conjugate of the ground literal is simply checked for previous occurrences.

An attractive benefit of HARP's two-layered architecture, with the base-level comprising the bookkeeping mechanisms such as the indexing scheme and the heuristic meta-level directing the actions of the base-level, is its capability to easily incorporate "admissible inference rules". The latter are derived rules of inference, i.e. they do not extend the range of what is provable without them but they make proofs more concise and more efficient. The formulation of these rules takes advantage of the control information that is available due to the richer syntax of first-order logic (as compared to the language of clauses). The behavior of the theorem prover can be specialized and made more human-like by using admissible inference rules corresponding to such familiar inference patterns as Modus Ponens, Modus Tollens, Modus Tollendo Ponens etc., as well as patterns that combine universal instantiation and Modus Ponens like the following: from $\forall x(Fx \Rightarrow Gx)$ and Fa infer Ga .

In Figure 5-1a we show a tree produced by the normal inference rules. The more readable and considerably leaner tree in Figure 5-1b results from an application of the admissible rule Modus Ponens.

Before an admissible inference rule can be applied, the matching (sub) formulas must be detected. The indexing scheme holds enough information to pinpoint possible matches.

Whenever a formula is entered into the tree, either initially as a premise or a conclusion or as a result of a decomposition operation, each predicate symbol in it is

linked to every unchecked formula in the tree that also contains this predicate symbol. Each new formula entered into the tree is checked for a matching counterpart. Since formulas are internally treated as nodes, flags can be easily attached to them. Thus this check is done efficiently by looping through the predicate symbols of the new formula to find the intersection of formulas which contains these predicates. Each formula considered as a candidate match is flagged accordingly. For example, consider $P \vee R$ in Figure 5-1b. Initially, the formulas where the predicate P occurs are flagged. On the next iteration, the formulas which contain R and which are also flagged to indicate the occurrence of P are selected as possible matching candidates.

Figure 5-1a

Figure 5-1b

Since all our formulas are in the language of first-order logic we cannot rely on unification to verify the candidate matches. Instead we simply pattern match the candidate pairs. Pattern matching compares data with a pattern. We choose the pattern to be the (sub) formula with the largest number of occurrences of universally quantified variables. In the case of a tie, an arbitrary selection is made. The universally quantified variables match any term. The other (sub) formula is treated as the data. For example, in Figure

5-2a $\forall xGx$ is chosen as the pattern since y is actually existentially quantified (due to the implicit negation introduced by the conditional). In Figure 5-2b neither Ga nor $\forall yGy$ match since the variable y is again existentially quantified.

$\forall xGx$
 $\forall yGy \Rightarrow P$
 P

Figure 5-2a

Ga
 $\forall yGy \Rightarrow P$
 $\neg \forall yGy$ P

Figure 5-2b

It should be noted that when a formula is entered into the tree its variables are also marked as existentially or universally quantified by recursively determining the sign polarity of its subformulas. To illustrate, the variable y in the formula $\neg(\forall xFx \Rightarrow \forall yGy)$ is marked as existentially quantified since the semantic evaluation rules cause the consequent of the conditional to be negated. For analogous reasons, x is marked as universally quantified. As a result, the type of variables within a formula is determined without search.

This mechanism which detects patterns of formulas can also be used to form specialized control regimes such as inheritance. For example, the following formulas Fa , $\forall x(Fx \Rightarrow Gx)$, $\forall x(Gx \Rightarrow \exists ySxy)$ are detected by a process similar to the matching described above and then the specialized control strategy is executed to derive $\exists yS(a,y)$. If indexing links which cluster the formulas were explicitly entered as part of the input (eg. as in frame systems), then the pattern matching process would not be necessary and the resulting behavior would be just as efficient as that exhibited by those special purpose representations which rely on such links to construct inferences. It should be noted that

in any case, even without user-provided links, HARP achieves specialized behavior whose efficiency is comparable, due to the matching procedure used, to that of the special purpose representation languages.

The basic algorithm is complete. (No proof of this fact will be given here.) Since the algorithm ensures that every applicable rule will eventually, if necessary, be applied to every unchecked compound, Smullyan's completeness proof [Smullyan68, p.57] carries over with straightforward but tedious modifications. However, as will be pointed out in section 5.3, some of our heuristics for detecting nontheorems do affect completeness.

5.2 Proof Condensation

In this section we will augment the basic tableau control structure with a proof condensation procedure. Like the heuristics described in the next section, this procedure makes proof construction more efficient and easier to follow. The proof condensation procedure complements the heuristics by compensating for possible shortcomings in the suggested advice. For ease and convenience of presentation, the examples discussed in this section will not utilize the full set of heuristics described in section 5.3.

The objective of the proof condensation procedure is to eliminate redundancies in the proof tree. It is called whenever a branch can be closed with a pair of conjugate literals. The algorithm detects and eliminates unnecessary branch points. During the process of proof condensation, the proof tree is shortened by the removal of these unnecessary branch points as shown in Figures 5-3, 5-4, and 5-5.

A branch point is a decomposed β node in the tableau tree. A node a is an ancestor of a node b if node b was produced by a series of decompositions from node a . A branch closure point (node) is created from two conjugate literals of the same branch. A node is an ancestor of a branch closure point if it is an ancestor of one of the conjugate

literals of the branch closure point. In this case we say that the branch closure point depends on that node. For example in the first tree of Figure 5-3 the β node $f \vee g$ is an ancestor of the closure point $(f, \neg f)$ in the leftmost branch. A branch point is necessary if and only if it is an ancestor of at least one of the conjugate literals of a branch closure point from each of its two subtrees.

The detection of unnecessary branch points relies on a bookkeeping mechanism which maintains reference counts associated with each branch point. The reference count determines whether that node is necessary for completing the proof and is maintained when open branches exist below the node. It denotes the number, in the current subtree, of dependent closure points for that branch point. If the reference count is zero after closing one of the subtrees of the branch point, then that branch point can be eliminated.

To facilitate the maintenance of these reference counts, each node records the β nodes it depends on. The memory requirements for these records is minimized by sharing these lists among the nodes. When a closure node is constructed, the reference counts of the β nodes that its two conjugate literals depend on are incremented by one. The dependency links of the conjugate literals in the closure node are traced to determine the set of branch points whose reference counts are to be incremented.

This algorithm exploits the ordered dyadic tree property of analytic tableaux (i.e. each branch point has a left subtree and a right subtree). Each β node contains a flag indicating the branch direction of that node. A left branch direction means that the β node is part of the left subtree of the previous β node. Similarly, a right branch direction means that the β node is part of the right subtree of the previous β node. This branch direction flag dictates the type of removals effected by the proof condensation procedure.

Descendant branches of a branch point are those created below that branch point. Two types of branch removals are possible: (A) removal of a branch point whose

descendant branches are all closed and (B) removal of a branch point with an open descendant branch. Both these removals shorten the resulting proof and thus yield a more transparent output. Type A removals are initiated from the closing of the rightmost branch of the subtree and do not immediately decrease the number of open branches left to close. But they increase the chances of other removals (both type A and type B) since reference counts are updated when unnecessary subtrees are eliminated. Type B removals do immediately decrease the amount of work left to be done but they do not increase the chances of further removals higher in the tree.

A type A branch point removal is always initiated from the right subtree of that branch point. It is triggered from previous proof condensation, closure point construction, or a combination of both. Removals are performed by the retraction of the corresponding β node decomposition choice.

Type A removal

Figure 5-3: The branch closure point of $(q, \neg q)$ does not depend on the β node $f \vee g$. Therefore this β node

can be eliminated. Since the subtree spawned by this β node has already been fully closed, this removal does not immediately reduce the number of open branches left to close. Notice that proof condensation could not have been performed when the leftmost branch was closed because the branch closure point $(f, \neg f)$ was a descendant of the β node $f \vee g$.

Type B removals are initiated from the left subtree by similar events (i.e. previous proof condensation, closure point construction, or a combination of both).

Type B removal

Figure 5-4: Again the branch closure point of $(q, \neg q)$ does not depend on the β node $f \vee g$. However in this case, removal of this branch point also causes the removal of an open branch which reduces the number of open branches left to close.

Whenever a type A removal is performed, the closure points of the left subtree of that branch point are no longer needed. The following figure (Figure 5-5) illustrates the retraction of dependencies emanating from closure points eliminated during a type A removal. These dependency retractions are reflected in reference count updates which could result in the identification of further redundancies (eg. see the subsequent type B

removal shown in Figure 5-5). The proof condensation procedure reshapes the proof tree to highlight additional unnecessary branch points.

Type A removal causing a Type B removal

Figure 5-5: The type A removal causes the descendants of the leftmost branch of the β node $\neg p \vee h$ to be eliminated. This elimination reduces the reference count of the β node $p \vee q$ which sets up the resulting type B removal.

We will now present a general outline of the proof condensation procedure, **Condense**. It is called with two parameters: 1) branch-direction which is the branch direction of the current subtree, and 2) branch-points which is an ordered list of the branch points in the current branch.

Condense (branch-direction, branch-points)

```

  if branch-points is not an empty list {i.e. non-nil}-
  then let br-pt be the most recent branch point {i.e. in lisp (car branch-points)}
      let new-branch-direction be the branch direction of br-pt
      if br-pt is redundant {i.e. if it has a reference count of 0}
      then retract the branch point br-pt
          if type A removal {i.e. branch-direction > 0}
          then remove the left subtree from the proof tree
          else undo changes since last branch point
          endif
      Condense (new-branch-direction, rest(branch-points))
  else if coming from the right {i.e. branch-direction > 0}
  then Condense (new-branch-direction, rest(branch-points))
  endif
endif
endif

```

End Condense

Because a decomposition operation may produce more than one literal, and since the same literal may be present several times on a single branch, several candidate closure points could exist for the same branch. For example, competing candidate closure points exist if the decomposition operation produces more than one literal, each of which has conjugate literals which already reside on the branch. Moreover there could be several different nodes for different occurrences of the same conjugate literal formula.

If there are several competing closure candidates, then the closure node that is chosen is the one that depends on the least number of branch points which previously had no dependents. This strategy facilitates the effective operation of the proof condensation procedure as shown by the transformation of the partially completed proof tree of Figure 5-6 to the complete proof of Figure 5-8.

Figure 5-6: The six possible closing nodes at the bottom of the above diagram (i.e. 1a - 2a, and 1b - 4b) are annotated with a label and a list of β nodes which they are dependent upon. For example, closing node 2b depends upon β_5 , β_3 , and β_2 while 1b depends only on β_5 . The difference is due to the two

occurrences of the $\neg V$ literal. Closing node 1b utilizes the $\neg V$ literal residing at the root of the tree while 2b uses the literal decomposed from the branch point β_3 which in turn was derived from β_2 . Both closing nodes incorporate the V literal taken from the branch point labelled β_5 .

Figure 5-6 displays the closing node choices for the last two branches of the partially constructed proof tree. If nodes 2a and 3b were chosen then proof condensation would not occur. Figure 5-7 shows a poor selection of closing nodes which prevents the Condense procedure from closing off the tree. The closing nodes actually chosen by the above strategy constitute the optimal choice in this case and enable the proof condensation procedure to transform the partially constructed proof tree of Figure 5-6 to the compact and completed proof tree of Figure 5-8.

Other strategies are also considered. One such strategy would favor closure points with dependencies closer to the tree root. Another such strategy would favor closure points that will introduce the smallest number of new dependencies on a set of distinguished predicates. This last strategy might be useful in algorithms searching for a minimally inconsistent set of predicates. (For instance, we would expect Reiter's algorithm for generating all possible diagnoses [Reiter86] to operate more efficiently with this strategy as it would reduce the number of new dependencies that would otherwise be created during the proof tree construction. A smaller set of inconsistent predicates constrains the resulting search which that algorithm undertakes.)

In summary, the proof condensation procedure is an efficient mechanism which compensates for some shortcomings in the set of heuristics. To illustrate this point clearly, the above examples do not apply the full set of heuristics that HARP normally uses. Indeed, the frequency of use of this condensation procedure is a measure of the pertinence of the heuristic advice. We rely on this measure to develop heuristics and to evaluate the performance of different sets of heuristics.

Figure 5-7: If closing nodes 2a and 4b were chosen then this diagram illustrates the tree produced by the proof condensation procedure. Notice that β_4 was the only branch point removed by the condensation procedure.

Figure 5-8: The closing nodes, 1a and 1b, actually selected by HARP are the optimal choice in this situation. The proof tree of Figure 5-6 is reshaped by the proof condensation procedure to produce the shortest possible proof.

5.3 HARP's Architecture and Heuristics

HARP's basic control structure, with the exception of proof condensation, is a modification of Smullyan's [Smullyan68] tree calculus and is thus amenable to a similar completeness proof. Although it can be proven that every inconsistent set has a closed tree, it is clearly not the case that every tree for an inconsistent set closes. (For instance, a branch might grow forever simply because of the indefinite repeatability of universal instantiation. To prevent this, the algorithm closes a branch as soon as a contradiction appears on it.)

But the basic control structure will often build needlessly branching trees. Our

heuristics are designed to prune trees without affecting HARP's completeness. (See the remarks below for some qualifications of this claim.) Most of our heuristic rules prioritize tasks, i.e. tasks more likely to lead to early closure are attempted sooner while heuristically less likely appearing tasks are only delayed rather than abandoned.

The heuristics fall into three groups: a) heuristics for efficient and human-like proof construction, b) heuristics for detecting nontheorems, i.e. for discovering as soon as possible when an open branch is complete, c) domain-specific heuristics.

We feel that a knowledge-based approach to heuristics, i.e. one which treats them as explicitly represented, independently modifiable rules, has several advantages over "hardwired" heuristics.

For one thing, an explicit, declarative representation of heuristics enables the use of knowledge engineering techniques such as the iterative development cycles successfully employed in other knowledge-based systems to capture judgemental pieces of advice. As is well known, a declarative knowledge base is easily modifiable. Modifiability is crucial, given the incremental nature of our enterprise. This is particularly apparent in the case of rules from groups b) and c): since logic is undecidable, new patterns detecting potentially infinite branches will have to be added tentatively and gradually; similarly, domain-specific heuristics have to be developed incrementally, as suggested by experience with inference patterns frequently occurring in certain applications.

Some further benefits result from a declarative formulation. For instance, it is easy to study interactions among heuristics. It is also straightforward to reformulate them for certain applications so as not to reorder but abandon attempted steps, thereby giving up on the goal of completeness in the interests of greater efficiency. In addition, the rules can be designed and entered at a level of conceptual detail that is comfortable for people.

Finally, heuristics are used in HARP to modify the basic control structure without

recoding and to experiment with different control regimes. Again, the insistence on an extensible and modifiable control structure suggests an explicitly represented knowledge base of heuristics. We will now outline an architecture that has been developed to fit these requirements.

The architecture consists of a base-level and a meta-level. The latter controls the base-level which executes actions, maintains bookkeeping records, and detects branch closures. Possible base-level actions include the decomposition of unchecked nodes and the application of admissible inference rules. The meta-level determines the most appropriate action to be performed and directs the base-level to carry it out. The base-level contains the proper procedures necessary to execute the action. The corresponding procedure is evaluated and then the indexing scheme and the tableau are updated to reflect the changes.

The meta-level functions in two stages: first, a set of applicable actions are formulated, and second, the heuristics described below are invoked to select the most suitable action. Currently, the n most well-connected nodes (where n is a parameter whose choice is left to the designer of the heuristics) are examined to produce this set of actions. The well-connectedness of a node is computed - from information in the indexing scheme - as the sum of the link strengths of all links entering that node. HARP only computes what is necessary to modify the information from the previous cycle. Thus the set of actions computed from the previous cycle is modified to reflect changes to the set of available nodes. The nodes, themselves, trigger the candidate actions. The node categories trigger possible decomposition rules and the formula patterns of these nodes may also trigger admissible inference rules.

Below we shall describe a few sample heuristics. With the exception of rules similar to the ones listed under H11 and H13, our heuristics fall into group a). No rules

in a) affect HARP's completeness property but some of the others may. Group c) rules are designed only with a particular application in mind and might be formulated to draw merely plausible but invalid inferences. Group b) rules apply in situations in which a set is consistent but the algorithm could never establish that fact conclusively. As an example, suppose we try to prove the invalid formula $\forall x \exists y Hxy \Rightarrow \exists y \forall x Hxy$. The proof takes the form of the following, potentially infinite branch:

$\forall x \exists y Hxy, \neg \exists y \forall x Hxy, \forall y \exists x \neg Hxy, \exists x \neg Hxa, \neg Hba, \exists x Hby, Hbc, \exists x \neg Hxc, \neg Hdc, \text{ etc.}$

Heuristics in b) keep track of this type of nonterminating loop and declare the above branch complete after a few cycles. This approach will, of course, not work in all cases but - in view of Church's theorem - no theorem prover can detect all nontheorems. It should be noted that successful proofs of theorems do not rely on any heuristics in b), i.e. the latter are only used to detect some types of nontheorems.

Heuristics are expressed in terms of the information provided by the indexing scheme. However, in order to avoid excessive formalism and to improve readability we shall present the sample heuristics in English. All of them are defeasible and are to be understood as carrying an implicit *ceteris-paribus* clause.

H0: Work on a branch until it closes or is known to remain open.

H0 partly specifies HARP's control structure by imposing a depth-first strategy which is appropriate given the goal of closing all branches. H0 is overruled when HARP is directed to establish the consistency of a set of assumptions. (The call to HARP, in this case, is $(\text{Consistent } (p_1 p_2 \dots p_n))$ and is defined as $(\text{Prove } (p_1 p_2 \dots p_{n-1}) \neg p_n)$. Prove, the top-level function, accepts as arguments a list of premises and a conclusion which it negates before creating any nodes.) The task of showing a set

consistent is prominent in several applications of HARP, e.g. in reason maintenance (where maximal consistent subsets of a given inconsistent set have to be found) and in plan detection (where the consistency of the known consequences of an action with the results expected under a plan has to be determined).

Establishing the consistency of a set of assumptions involves detecting an open yet complete branch and is accomplished faster, on the average, when HARP is explicitly told to override H0 and work breadth-first instead.

H1: Work on a compound until literals are reached.

Once HARP has decided to decompose a formula, it prefers to continue working on that formula's compound components (if any), thus assuring that the preconditions for closing a branch or declaring it open are achieved without delay. H1 is overruled when admissible rule patterns are found.

H2: Favor rules that introduce as few new nodes as possible.

H2 is an obvious rule that prevents unnecessarily branching trees and unnecessarily long branches by postponing, as long as appropriate, the decomposition of β and γ formulas. H2 also favors applications of UI that lead to α -instances.

H3: Prefer existential to universal instantiation.

H3 reflects the fact that existential quantifiers, unlike universal ones, are dropped only once. H3 is overridden only when the available indexing information indicates that an universal instantiation would produce a contradiction immediately, as in the following example: $Fa, \forall x \neg Fx, \exists y Gy$.

H4: Favor formulas with embedded existential quantifiers.

To apply H3 it is often necessary to get at embedded existential quantifiers. Relying on its indexing scheme, HARP considers only those quantifiers that will end up as existential after the decompositions required to bring them to the surface have been applied. Thus, $\neg(Fa \wedge \exists y Gy)$ does not but $\forall x(Fx \Rightarrow \neg \forall y \neg Gy)$ does contain an existential quantification. For example, given $\forall x(Fx \Rightarrow \forall y(Gy \Rightarrow \neg Lxy))$ and $\forall x(Fx \Rightarrow \exists y(Hy \wedge Lxy))$, H4 recommends working with the second formula first.

As far as UI is concerned, H4 includes the following advice: given a previously uninstantiated universal quantification, that has a dependent formula with a corresponding existential quantification, not more than one β -split away, then prefer the dependent formula over the universal one.

Consider the following example (Figure 5-9), in which H4, by advising UI on 2) before 1), saves an otherwise necessary extra instantiation of 2) and a few nodes:

Figure 5-9

H5: Favor compounds derived from the negation of the conclusion.

H5 corresponds to a tendency to conform to the set-of-support strategy and aids in selecting syntactically relevant premises. However, like other heuristics, H5 only reorders the priority queue and, thus, does not rigidly impose the set-of-support strategy. For example, H5 does not prevent the derivation of R from the inconsistent set $P, P \Rightarrow Q, \neg Q$.

H6: Avoid clearly useless work.

H6 comprises several work reducing rules like the following two:

if a propositional variable occurs only once in the premises and not in the conclusion and, hence, cannot possibly contribute to a proof, it is thrown away;

if $\neg$ occurs nowhere in the root set then no proof is attempted because at least one branch is sure to be open.

H7: Choose carefully among branching compounds.

According to H7 a conditional is decomposed only if either its antecedent or the negation of its consequent or, preferably, both appear as literals higher up on the same branch. Similar rules apply to the other β formulas. In short, H7 enjoins to break down the β formula that maximizes the chance of closures. The following example illustrates how selective H7 is.

Suppose we wish to prove $P \Rightarrow Q$, given these premises: 1) $R \Rightarrow S$, 2) $\neg P \Rightarrow T$, 3) $\neg T \Rightarrow \neg Q$, 4) $U \Rightarrow T$, 5) $U \Rightarrow \neg P$, 6) $\neg Q \Rightarrow U$. Because of H5 and H7, HARP begins with the desired conclusion, then skips 1) because it is clearly irrelevant to the entire proof, 2) and 3) because, though syntactically relevant, neither will clash

with an available literal, and 4) because it is irrelevant given the branch so far. 5) and 6) each produce one closure so that either one can be chosen next. Eventually, HARP builds the tree of Figure 5-10:

Figure 5-10

H8: Favor fresh universal quantifications.

For each universal quantifier a record is kept of how often it has been used and with what instantiating terms. Less frequently used quantifiers are preferred. H8 reflects the empirical observation that usually in AI applications each relevant premise makes a definite and unique contribution to a proof. H8 only delays multiple uses of UI but does not proscribe them in all circumstances. In the following example, nothing remains to do after 5) but to reuse UI on 1).

- 1) $\forall x \exists y \neg(Fy \Rightarrow Fx)$, 2) $\exists y \neg(Fy \Rightarrow Fa)$, 3) $\neg(Fb \Rightarrow Fa)$, 4) Fb , 5) $\neg Fa$,
 6) $\exists y \neg(Fy \Rightarrow Fb)$, 7) $\neg(Fc \Rightarrow Fb)$, 8) Fc , 9) $\neg Fb$ *.

H9: Minimize the introduction of new parameters.

H9 suggests that universal instantiations be done only with respect to terms already on a branch unless there are none yet. Moreover, for the terms on the branch, only those instantiations are made that are sanctioned by the other heuristics. The proof below

2 of/de 2

MicroD

(Figure 5-11) illustrates H9 and some previous heuristics.

1)	$\forall x \exists y Fyx$	<i>premise</i>
2)	$\forall x \forall y (Fxy \Rightarrow Gxy)$	<i>premise</i>
3)	$\neg \forall x \exists y Gyx$	<i>negation of conclusion</i>
4)	$\exists x \forall y \neg Gyx$	<i>quantifier negation from 3)</i>
5)	$\forall y \neg Gya$	<i>EI from 4), via H3</i>
5')	$\neg Gaa$	<i>useless, discouraged by H4</i>
5'')	$\neg Gba$	<i>discouraged by H9 because of the new parameter</i>
6)	$\exists y Fya$	<i>UI from 1), not 2) via H4, H2</i>
7)	Fba	<i>EI from 6), via H3</i>
8)	$\neg Gba$	<i>UI from 5), not 2), via H2</i>
9)	$Fba \Rightarrow Gba$	<i>UI from 2)</i>
9')	$Faa \Rightarrow Gaa$	<i>9'), 9''), 9''') are not created because they do not advance the proof</i>
9'')	$Fab \Rightarrow Gab$	
9''')	$Fbb \Rightarrow Gbb$	
10)	$\neg Fba$	
	Gba	
	*	*

Figure 5-11

H10: Apply admissible rules whenever possible.

Our mechanism to implement admissible rules is explained in section 5.1. Admissible rules simplify proofs while not affecting the derivability relation: whatever is derivable with their help can also be derived without them. They have several important advantages, however, including considerable gains in efficiency, conciseness and readability and a greater similarity with the basic inference patterns of natural deduction systems. While in principle any theorem could be attached to any open branch without

affecting semantic adequacy, we have currently implemented only Modus Ponens, Modus Tollens, Modus Tollendo Ponens and patterns combining UI and Modus Ponens. The two next examples (Figures 5-12 and 5-13) show the work saving and clarity enhancing effects of H10 (on the right).

Figure 5-12

Figure 5-13

H11: Apply domain-specific rules whenever possible.

H11 comprises the rules from group c) and instructs HARP to augment a given premise set with any available pieces of semantic and world knowledge before embarking on a proof. These added domain-specific rules are attached to individual predicates and thus create no overhead in general. But they play an extremely important role in making HARP a more useful tool in practical applications.

Figure 5-14

One advantage gained from using domain-specific rules has to do with the fact that most acceptable arguments actually encountered are enthymematic. Such an argument is invalid as it stands but may become valid when unstated background information is supplied. H11 allows the user to state such information only once - when it is associated with a specific predicate -, instead of having to remember and repeat it for each argument that implicitly appeals to it. For instance, certain two-place predicates might be marked as transitive. When such a predicate occurs in a formula, the base-level automatically augments the explicitly given premises with an appropriate specialization of the general transitivity pattern. For the next example, we have marked 'taller' as transitive and as the converse of 'shorter'. This, together with H11, enables HARP to derive 'Joe is taller than Jim' from 'Joe is taller than Jane' and 'Jim is shorter than Jane' (see Figure 5-14). Note that this intuitively valid argument is clearly invalid as stated.

H12: Identify complete open branches as quickly as possible.

H12 is intended to determine quickly whether the root set of an infinite tree is consistent. If an open branch contains no unchecked compounds except universal quantifications, and if every such quantification is instantiated with respect to all and only the terms above it on that branch then the branch may be declared complete. The next two trees (Figure 5-15) show how certain types of nontheorems are detected rapidly and reliably. The left one proves that $\exists x(Fx \wedge Gx)$ and $\exists x(Gx \wedge Hx)$ do not imply $\exists x(Fx \wedge Hx)$ and the right one proves that $\forall x(Fx \Rightarrow \exists yGxy)$, $\exists xFx$ and $\exists x\exists yGxy$ do not imply $\forall x\exists yGxy$. In the case of both trees further applications of UI would be pointless.

Figure 5-15

H13: Watch out for nonconverging $\forall\exists$ -patterns.

H13, like H12, attempts to identify nontheorems as quickly as possible. Its success, however, unlike that of H12, in pinpointing nontheorems is not guaranteed. After all, a complete proof procedure - unlike a decision procedure - does not, in general, give negative answers to validity questions. The intention behind H13 is to identify - sometimes and fallibly - certain common types of nontheorems that would go undetected by the basic algorithm and H12.

H13 consists of several rules from group b) that are invoked whenever at least one universal quantifier precedes at least one existential quantifier and universal instantiation is reapplied to new constants introduced by existential instantiation. The quantifiers do not have to form a consecutive string: any formula in which a universal quantifier might repeatedly bring an existential one to the surface activates the various rules of H13. An

additional condition for triggering H13 is that the scope of the quantifiers be polyadic. There are two reasons for this further condition. First, in the monadic case, rules of passage, i.e. rules for moving quantifiers across truth-functional connectives, could be applied to purify or pull apart any string of quantifiers such that each quantifier governs a truth-function of terms involving only the variable of quantification. Such purified monadic formulas could then be checked for nontheoremhood by our other heuristics. Second, monadic predicate logic is decidable anyhow, at least in-principle.

The various rules of H13 recognize particular $\forall\exists$ -patterns that we have empirically identified as frequently indicating infinite branches. Whenever such a pattern is matched, its branch is immediately declared complete. The basic nonclosing $\forall\exists$ -pattern, of which all the other currently detected patterns are variants, is this:

$\forall x\exists y Fxy, \exists y Fay, Fab, \exists y Fby, Fbc, \exists y Fcy, Fcd$, etc.

At this point, H13 decides that it is pointless to continue the branch.

It should be noted that the rules in H13 do not, of course, declare a branch complete as soon as an $\forall\exists$ -pattern is found. After all, not all $\forall\exists$ -patterns are nontheorems (see the theorem below), just as many nontheorems are not $\forall\exists$ -patterns (see the examples for H12). Instead, when an $\forall\exists$ -pattern is found, each rule of H13 begins to look in the current branch for a particular pattern of instantiations that is likely to continue indefinitely. If such a pattern occurs then the branch is declared complete, otherwise these rules have no effect on the proof. As an example, consider the proof of the following theorem:

$\exists x\forall y (Fay \Rightarrow Fxy)$.

$\neg\exists x\forall y (Fay \Rightarrow Fxy), \forall x\exists y\neg(Fay \Rightarrow Fxy)$ (at this point, the rules in H13 are activated and begin their search for nonclosing patterns), $\exists y\neg(Fay \Rightarrow Fay), \neg(Fab \Rightarrow Fab), Fab, \neg Fab$ *. Since the rules have not found a nonconverging pattern of

instantiations, the branch closes without their interference.

Infinite branches seem always to be due to reapplications of UI to new terms introduced by EI. By the same token, if no existential quantifier is preceded by a universal one then the process of instantiation must come to an end. In this latter case, the number of instances for any premise is at most equal to the product of the number of universal quantifiers in that premise and the sum of the existentially quantified variables and constants in all premises.

A typical pattern recognized as nonclosing by H13 follows (see Figure 5-16).

1)	$\forall x \exists y Fxy$	<i>premise</i>
2)	$\exists x \exists y \neg Fxy$	<i>premise</i>
3)	$\neg \exists x \forall y Fxy$	<i>negation of conclusion</i>
4)	$\forall x \exists y \neg Fxy$	<i>quantifier negation from 3)</i>
5)	$\neg Fab$	<i>EI from 2)</i>
6)	$\exists y \neg Fay$	<i>UI from 4)</i>
7)	$\neg Fac$	<i>EI from 6)</i>
8)	$\exists y Fay$	<i>UI from 1)</i>
9)	Fad	<i>EI from 8)</i>
10)	$\exists y \neg Fby$	<i>UI from 4)</i>
11)	$\neg Fbe$	<i>EI from 10)</i>
12)	$\exists y Fby$	<i>UI from 1)</i>
13)	Fbf	<i>EI from 12)</i>

At this point, the pattern 7), 9), 11), and 13), together with the repetition of UI-EI moves from 4) and 1) is recognized as nonterminating and the branch is declared open. Similar special instances of recognized nonterminating patterns are $\neg Fab$, Fac , $\neg Fae$, Fad or $\neg Fab$, Fbc , $\neg Fcd$, Fde .

Figure 5-16

This concludes our description of the currently implemented heuristics and the

supporting architecture of HARP.

These last two examples (Figures 5-17 and 5-18) are included to illustrate how elegant HARP's proofs are when all our heuristics are turned on.

Figure 17

Figure 5-18

6 THE IMPLEMENTATION

In this chapter, implementation details and empirical results are treated. The primary topic is the implementation of the program used to produce the timing and benchmark results stated in section 6.3. This cleverly implemented version is the result of compiling many of the heuristic strategies of section 5.3. It is of interest to see how these general heuristic strategies were compiled into an efficient procedural format. The techniques and data structures of the optimized program with the hardwired heuristics will be discussed to illustrate the many details and concerns of constructing an efficient full-scale theorem prover.

The implementation language and development environment and how they have impacted upon the final system will be discussed in section 6.1. The underlying data structures of the tableau and the indexing scheme are described in section 6.2.2. We will also outline the algorithms which update the proof tree and maintain the proper problem solving state especially when backtracking is triggered from a branch closure (section 6.2.3).

6.1 Implementation Language and Development Environment

HARP is implemented in approximately 3000 lines of ZetaLisp code on a Symbolics 3600 Lisp Machine running release 5.0 of the operating system. ZetaLisp is a descendant of MacLisp and is the language that forms the backbone of the Symbolics Lisp Machine. In addition to the basic theorem prover, there also exist routines to interactively trace the proof construction and print out the final proof tree (see the Appendix for samples of proof tree displays).

In retrospect, Lisp is a good choice for the implementation of a program that is as complex as this one. The complicated algorithms require intricate operations on data

structures that need to be efficient, both in terms of time and space. Yet since the development of HARP was conducted in a rapid prototyping fashion, a high level language was needed to maintain a large degree of transparency and modifiability. We did not wish to become bogged down with irrelevant implementation details. Lisp is a language that satisfies these criteria.

The symbol manipulation facilities of Lisp (atoms and property lists) were especially useful for the representation of the logical formulas and the implementation of the indexing scheme. To illustrate, predicates are represented by atoms and the indexing scheme through tables based on property lists.

The dynamic memory allocation feature of Lisp simplified the algorithm which maintained the tableau. Automatic garbage collection eliminated the complicated and messy requirement of writing code to keep track of memory cells. The built-in garbage collector is automatically invoked by the Lisp system to reclaim memory.

In addition, we feel that the development and debugging environment of Lisp is unsurpassed; code as efficient as that of any other language is more quickly written and debugged. These features are crucial to the successful construction of a powerful theorem prover.

Lisp is both an interpreted and compiled language. The interpreter is a good platform for incremental development and testing, while the compiler is necessary for efficient program execution.

The interactive debugger and data structure inspector of the the Lisp machine environment was crucial to the successful development of a program which is as large and complex as this one. The debugger is completely integrated with the rest of the development environment. From the Lisp execution window (i.e. the Lisp Listener), it is invoked by error conditions and breakpoints. Once inside the debugger, the user may

scroll through the activation stack frame, examine the variables and data structures, change the values of variables, modify the program, and resume the execution of the program at any arbitrary point. As the user is scrolling through the execution history trace, the desired modification of any of the functions is initiated by a one step control sequence which results in the invocation of the full screen editor on the function. The editor is automatically positioned in the appropriate location of the corresponding source file. After the modifications are made, an incremental compile or load of only the changes will prepare the system to resume execution with the modifications in effect.

The inspector facility may be invoked at any point to provide a more sophisticated viewing mechanism for Lisp objects and data structures. Destructive operations can be performed on these structures. Since Lisp code is in the form of the list data structure, the code may also be examined and changed through this data structure inspector facility.

The advise and trace utilities were extremely helpful in debugging. Arbitrary trace and debug statements can be inserted into the program without modifying the original code. The original source code is not cluttered or unnecessarily complicated with extraneous code. Different aspects of the program can be monitored by turning on switches. When efficiency is desired, the statements are removed by simply turning off the switches. This feature is extremely convenient during the intensive development phase when the program code is in a state of flux.

To illustrate, when code is being modified, previous portions of the program which were once thought to be functioning correctly may now be displaying anomalous behavior. In this case, the old trace and debug statements may be turned on to monitor that portion of the program. Once the problem is resolved, these trace and debug statements can be removed by simply flipping the switch again. This debugging code is always available for execution but it never clutters up the original source.

In summary, we feel that the selection of Lisp as the implementation language was a good decision and a significant factor in the successful development of the software. In the succeeding exposition, a basic knowledge of Lisp is assumed.

6.2 Program Details

When invoked, the system first initializes its proof state. In particular, the variable *br-num*, which denotes the current branch being processed, is set to 1. This variable is used as a label during the process of recording changes to the proof state. Records of changes are maintained so that restoration may be performed when backtracking is initiated from branch closure.

The input to the theorem prover consists of a list of premises and the conclusion which is to be proved. Each of the premises along with the negation of the conclusion is entered into the tableau by *categorizing* the well-formed formula and *adding* it to the indexing scheme. During this process, each formula is given an internal form which we shall often refer to as a node. Each node is actually a record structure (described below) containing system information, such as its position in the tableau. Once the initial formulas are entered into the tableau, the main loop is invoked and continues to run until either the theorem is proven or until it is discovered to be a nontheorem.

As the main loop executes, new branches are added, branches are closed, proof condensation reshapes the tree, and the indexing scheme is maintained to properly reflect changes due to decomposition and backtracking. The sophisticated software machinery which handle these tasks will be discussed in the succeeding sections.

6.2.1 Categorizing the Formula

The set of all the well-formed formulas are broken down into the following

mutually exclusive categories: *double-negation*, *and*, *not-or*, *not-conditional*, *existential*, *not-existential*, *not-universal*, *atomic-formula*, *not-atomic-formula*, *not-and*, *or*, *iff*, *not-iff*, *conditional*, and *universal*. To illustrate, let Φ_1 and Φ_2 be any well-formed formulas. A formula belonging to the *double-negation* category is of the form $\neg(\neg(\Phi_1))$; the *not-conditional*, $\neg(\Phi_1 \Rightarrow \Phi_2)$; the *not-universal*, $\neg(\forall x \Phi_1)$; *not-and*, $\neg(\Phi_1 \wedge \Phi_2)$; *or*, $\Phi_1 \vee \Phi_2$; and the *existential*, $\exists x \Phi_1$. These categories are utilized by the program to select the Lisp function which performs the actual decomposition.

Similar to the α , β , γ , and δ breakdown of well-formed formulas presented in section 4.2, the categories are clustered together to form larger classes of formulas which we will denote by the *type* of the formula. There are three types of formulas recognized by the program: *alpha*, *beta*, and, *universal*. These types have a great deal of effect on the node order in the decomposition agenda.

The *alpha*-type formulas are the usual α formulas (i.e. *double-negation*, *and*, *not-or*, *not-conditional*) plus the *existential* (δ type formulas), *not-existential*, and *not-universal* formulas. The *beta* variety is composed of the *not-and*, *or*, *iff*, *not-iff*, and *conditional* categories. The *universal* type consists of just the universally quantified formulas (the *universal* category). In the implementation, the type of each category is found on the property list of the symbolic atom representing that category. For example, *beta* is the value of the *type* property of the *not-and* atom. Thus the formulas need only be categorized once, whereupon the type can be determined via a one step operation.

6.2.2 The Indexing Scheme

Conceptually, the indexing scheme is a graph-like structure with links between conjugate literals of the formulas in the tableau. When a formula is *added* to the indexing scheme, all of its literals (gathered by recursively scanning the subformulas) are marked

by location records. Each record is accessed by a key consisting of the predicate, sign polarity, and *canonicalized* arguments of the literal. The canonical form of the arguments of a literal is a list whose length is equal to the number of arguments of the predicate in the literal. The *i*th element of the list corresponds to the *i*th argument of the predicate. If the argument is a universally quantified variable then the list element is the symbol *U*; if the argument is an existentially quantified variable, the list element is a pointer to the enclosing existentially quantified formula scoping that variable; and finally, if the predicate argument is a constant then the list element is also that constant.

In the program each predicate is a symbol atom with two properties, one for each sign polarity. The value of each property is a table where the keys are the canonicalized arguments of the predicates with that polarity. These keys access the location records of those literals conforming to the key. Every literal is represented by a location record whose declaration follows:

```
(defstruct (location (:type :array)
                    :named
                    :copier
                    :predicate
                    :conc-name)
  (node nil)
  (actual-args nil)
  (number-of-splits nil)
  (beta-path nil))
```

The `(:type :array)` option specifies that the record structure is to be implemented internally as an array. It is a named structure which means that it has an extra information field containing its name and record type. This information is used to answer print messages sent to the structure when, for example, the Lisp `describe` command is invoked

upon it. The `:predicate` and `:copier` keywords automatically generate functions which can, respectively, detect records of this type and copy records of this type. The `:conc-name` keyword specifies that the names of the slot access functions be the concatenation of the name of the record type (i.e. in this particular example, *location*) and the slot name.

Other than the internal system specifications discussed above, this `defstruct` declaration assigns four slots to a location record. The *node* slot points to the node which represents the formula where the literal resides. The *actual-args* slot points to the arguments of the predicate in the literal. The *number-of-splits* slot contains the number of beta decomposition that are necessary to reach the literal. If this number is two or less then the *beta-path* slot holds a list with at most two elements consisting of any combination of the symbols *left* and *right*. The symbol *left* designates the left branch of a beta decomposition while the symbol *right* denotes the right branch. Both the *number-of-splits* and the *beta-path* slots are important pieces of information which determine the strength value of any conjugate link formed with this literal. Eventually this information is utilized to order the node decomposition agenda.

Links between conjugate literals are maintained by accessing this database of records. A matching function is called to correlate potential candidates. The potential candidates are the location records in the table on the property list of the predicate.

For example, suppose we wanted to find the conjugates of the literal signified by the bold characters in the following formula: $\forall x((\neg(\exists y(P(a,x,y))) \Rightarrow (Q(x)))$, where, notice, *x* is a universally quantified variable, *y* is an existentially quantified variable, and *a* is a constant. This means the locations of the negated predicate *P* are desired. This table of locations is found by accessing the negative polarity property of the symbolic atom *P*. The list of locations is narrowed considerably by applying a matching procedure to pinpoint those locations whose canonicalized predicate arguments seem to correspond

to the ones in question, i.e. $(a \cup (\exists y(P(a,x,y))))$).

The matching function compares the canonical form with the key of every entry in the table. It iterates through the two lists comparing each of the corresponding pair of elements via the following code:

```
(loop for element1 in args
      for element2 in key
      (if (neq element1 element2)
          (return nil))
      finally (return t))
```

If the iteration reaches the end of both lists at the same time and without being terminated prematurely by the above return statement, then the match is successful. As the reader can see, this match procedure is very efficient as it just compares two memory references in each iteration.

We will often refer to this process as *adding* a formula into the indexing scheme. Literal location records are created and links between conjugate literals are formed. The information supplied by the indexing scheme is exploited to order the nodes on the decomposition agenda (see also section 5.1). As a result, the order of nodes in the decomposition agenda may change. For formulas not yet on the agenda, if, during the process of adding a formula to the indexing scheme, a link is created to a conjugate literal, then the node for that formula is inserted into the agenda. Since this is the only way a node can reach the agenda, formulas which are completely irrelevant to the proposed theorem are ignored.

It should be mentioned, during this process, if the formula itself is a literal (i.e. either an atomic formula or the negation of an atomic formula), then a special function called *process-literal* is invoked to determine whether the branch can be closed. This

function simply checks the database to see if there are any formulas which themselves are the conjugate of that literal.

6.2.3 The Main Loop

Once the initial formulas comprising of the premises and the negation of the conclusion are entered into the tableau, the main loop is invoked. The main/top-level loop of HARP (see also section 5.1) iterates until either the theorem has been proved or until it is discovered to be a nontheorem. At each iteration stage, a node is selected for decomposition. The type of the node (i.e. *alpha*, *beta*, or *universal*) determines the function that is executed to supervise the desired decomposition.

If it is an *alpha*-type node then the function *process-alpha* is called. *Process-alpha*, first, removes the *alpha* formula from the indexing scheme by eliminating the location records of each of its resident literals. This is a scanning process which is recursively performed on the subformulas of the original formula and can be considered the reverse operation of adding the formula into the indexing scheme. Once all traces of the formula are removed, the category triggers the selection of the specific function which is called to perform the actual decomposition operation that results in the descendant nodes. Each of the new formulas is then entered into the tableau in a fashion similar to the undertaking performed on the original input formulas.

The function *process-beta* is called to handle the decomposition process of a *beta* node. Like *process-alpha*, the node is first removed from the agenda and indexing scheme. The node category determines the function performing the primitive decomposition operation that yields two lists of descendant nodes. A branch point is created and the variable *br-num* is incremented by 1. The first list of nodes is entered into the indexing scheme while the second list is stored with that branch point. When the left

branch is closed off, backtracking will return the main loop to the newly created right branch; after proof condensation is executed and the proof state restored to that in effect when the branch point was created, this second list of nodes is entered into the tableau.

Universal nodes need a constant for universal instantiation when they are processed for decomposition by the function *process-universal*. These formulas always remain in the indexing scheme since their literal location records are never removed. The node is often left on the agenda, so that they may be decomposed an indefinite number of times. However, their priority in the decomposition agenda is dependent upon the number of constants (in matching conjugate literals) which correspond to the variable quantified by the *universal* formula. Each time a decomposition is performed, a new constant from this set is offered, and the priority of the *universal* node is decreased accordingly. The node is removed from the agenda, only when there are no new constants left for instantiation.

We will now discuss branch points and the update operations performed on the tableau when a branch is closed.

A branch point is a record structure used to store pieces of information such as the newly created nodes of the right branch and the journal of changes to the proof state (i.e. changes to the indexing scheme, proof tree, and agenda). When a branch point record is created, its information slots are initialized and the record is pushed onto a stack of branch points, pointed to by the variable *br-pt*.

Whenever a branch is closed the *br-pt* stack is popped to find the most recent branch point. The variable *br-num* is decremented by one, and the proof state is restored to the state in existence when the branch point was created.

The restoration is accomplished by unraveling all the changes which affected the original state. Each value in the proof state is recorded as a pair where the first element is the *br-num* indicating when this value was assigned. The second element of the pair is

the actual value. Every function which changes the proof state calls the function *observe-change*. If the current *br-num* is different from the *br-num* of the old value then the old value pair is *consed* (Lisp terminology for adding elements to a list) onto a list pointed to by the variable *changes*.

Once the proof state is restored, the variable *changes* is set to the value of the *changes* slot of the branch point. This slot contains the old value of the *changes* variable in force when the branch point was created. Additional modifications of the proof state due to new decompositions are *consed* onto this list. The stage is now set for the next round of backtracking.

The entire loop centers around this branch point stack, pushing and popping it, and maintaining the proper proof state until the completion of the proof. This stack is a central data structure in the program.

The tableau is implemented as a tree of nodes where each node represents a logical formula. For example, the initial tableau would consist of a single branch of nodes containing the premises and the negation of the conclusion. As the proof proceeded, the tableau would branch into a more tree-like structure. In order to capture this form, each node is defined as a record structure with slots for pointers to its ancestral node and descendant nodes.

Special information is also attached to the node if it is a *beta* type that was responsible for the creation of a branch point. In this case, the node has a counter attached to record a reference count on the number of dependent closure nodes (remember the proof condensation procedure requires this information; see section 5.2).

The proof condensation procedure is called whenever a branch is closed. This procedure fits in quite elegantly with the rest of the proof procedure. In fact, the entire condensation algorithm is implemented with four functions in less than 30 lines of code.

These functions, however, rely on other machinery to update reference counts when a branch is closed.

After the proof condensation procedure returns, the branch path is copied so that it will be available for display in the final proof tree. It is necessary to copy the entire branch because, as previously mentioned, the entire proof state (including the tableau) is restored when the main loop moves to the next tableau branch. Of course, copying the entire branch after every closure is very inefficient; it is probably only necessary to copy the branch up to the last branch point (it was just easier to write a procedure to copy the whole thing). In any case, in situations where the display of the proof tree is not necessary (such as when HARP is utilized as a pure deductive engine), this feature could be disabled.

There also exists a routine to print out the proof tree in a sideways manner. This routine relies on these branch copies. Sample output and a description of these proof trees are presented in the Appendix.

6.3 Empirical Results

In this section we will discuss the solution of the challenge problem Schubert's Steamroller. This test problem is often used as a benchmark to compare different theorem provers [Walther84, Stickel86]. The English version of the problem is the following statement:

Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while

birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore there is an animal which likes to eat a grain-eating animal.

The following predicates capture these relationships:

$A(x)$ - x is an animal	$W(x)$ - x is an animal
$F(x)$ - x is a fox	$B(x)$ - x is a bird
$C(x)$ - x is a caterpillar	$S(x)$ - x is snail
$G(x)$ - x is a grain	$P(x)$ - x is a plant
$M(x,y)$ - x is much smaller than y	$Eats(x,y)$ - x likes to eat y

The first-order logic formulation of the premises of the problem is the following:

- 1) $\exists x (W(x))$
- 2) $\exists x (F(x))$
- 3) $\exists x (B(x))$
- 4) $\exists x (C(x))$
- 5) $\exists x (S(x))$
- 6) $\exists x (G(x))$
- 7) $\forall x (W(x) \Rightarrow A(x))$
- 8) $\forall x (F(x) \Rightarrow A(x))$
- 9) $\forall x (B(x) \Rightarrow A(x))$
- 10) $\forall x (C(x) \Rightarrow A(x))$
- 11) $\forall x (S(x) \Rightarrow A(x))$
- 12) $\forall x (G(x) \Rightarrow P(x))$
- 13) $\forall x (A(x) \Rightarrow ((\forall y (P(y) \Rightarrow Eats(x,y))) \vee (\forall y (A(y) \wedge M(y,x) \wedge (\exists z (P(z) \wedge Eats(y,z))) \Rightarrow Eats(x,y))))))$
- 14) $\forall x,y (C(x) \wedge B(y) \Rightarrow M(x,y))$
- 15) $\forall x,y (S(x) \wedge B(y) \Rightarrow M(x,y))$
- 16) $\forall x,y (B(x) \wedge F(y) \Rightarrow M(x,y))$
- 17) $\forall x,y (F(x) \wedge W(y) \Rightarrow M(x,y))$
- 18) $\forall x (W(x) \Rightarrow (\forall y (F(y) \Rightarrow \neg Eats(x,y)) \vee (\forall y (G(y) \Rightarrow \neg Eats(x,y))))$
- 19) $\forall x (B(x) \Rightarrow (\forall y (C(y) \Rightarrow Eats(x,y)) \wedge (\forall y (S(y) \Rightarrow \neg Eats(x,y))))$
- 20) $\forall x (C(x) \Rightarrow (\exists y (P(y) \wedge Eats(x,y)))$
- 21) $\forall x (S(x) \Rightarrow (\exists y (P(y) \wedge Eats(x,y)))$

The conclusion of the problem differs among various authors. Some authors (eg.

[Pelletier82]) take the phrase 'grain-eating animal' to mean an animal that likes to eat *some* grain (denote this interpretation by *). Walther [Walther85] interprets this phrase to mean an animal that likes to eat *all* grains. Since we have first encountered this problem in [Walther85] we will adhere to his interpretation. However, he differs in the interpretation of this challenge problem within even the same paper. His two different interpretations of the conclusion are the following:

$$\begin{aligned} 22) \exists x, y (A(x) \wedge A(y) \wedge (\forall z (G(z) \Rightarrow \text{Eats}(x, y) \wedge \text{Eats}(y, z))) & \quad ** \\ 22') \exists x, y (A(x) \wedge A(y) \wedge \text{Eats}(x, y) \wedge (\forall z (G(z) \Rightarrow \text{Eats}(y, z))) & \quad *** \end{aligned}$$

These two interpretations are not logically equivalent. The interpretation ** implies the interpretation *** and not vice versa. In other words, a proof of ** will suffice for *** while conversely a proof of *** is not sufficient for **. However, with the assumption of the nonemptiness of the grain sort (premise 6), the two conclusions are logically equivalent. (see section 6.2 which will discuss the underlying reason which led to the creation of these two interpretations). Both ** and *** imply a solution to *. HARP is able to solve both interpretations of this problem in roughly the same time frame. The resulting proofs are both of the same approximate size and structure.

Regardless of which interpretation is taken, the solution expressed in English is the following:

"Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Snails are much smaller than birds and like to eat some plants. Because birds do not like to eat snails, it must be the case that birds like to eat all plants, including grains. Wolves do not like to eat grains. Therefore, they must like to eat all animals much smaller than themselves that like to eat some plants. Because foxes are much smaller than wolves and wolves do not like to eat foxes, it must be the case that foxes do not like to eat plants. Therefore, foxes like to eat

birds, since birds are grain-eating animals that are much smaller than foxes." [Stickel86, p. 92]

Even though we realize that the solution time to this challenge problem is highly dependent upon the underlying machine and implementation, these results can and are still used as an approximate benchmark for comparison between different theorem provers (eg. see [Walther85, Stickel86]). HARP is able to solve ** in 14 seconds. The resulting tableau has 48 closure points. The solution of *** is achieved in 20 seconds and the proof tree contains 49 closing nodes. Screen dumps of these proof trees can be found in the Appendix.

Theorem Prover	Problem	Time (sec)
McCune/UR	*	158
McCune/LUR	*	44
McCune/UR	**	288
McCune/LUR	**	96
Lusk & Overbeek/UR	***	340
Lusk & Overbeek/QUR	***	155
McCune/UR	***	660
McCune/LUR	***	120
Plaisted/LOCK	***	6
Walther/MKRP	***	262

Figure 6-1: Solution times for Schubert's Steamroller

For the reader's information, Figure 6-1 displays the solution times of some of the other efforts devoted to the clausal translation of Schubert's Steamroller (from [Stickel86]). Again, we must re-emphasize that these times are provided only to indicate

the tremendous complexity of Schubert's Steamroller. This table should reinforce the reader's appreciation of the significance of HARP's timely solution to the original first-order version.

It should be noted that Plaisted's solution was based on locking resolution where problem dependent numerical indices are assigned to each literal [Stickel86]. McCune, Lusk and Overbeek employed many-sorted logic approaches to achieve their solutions. A brief description of these alternative methods can also be found in [Stickel86, Walther85, Cohn85].

7 MANY-SORTED LOGIC

In this chapter we will discuss many-sorted logic and its application to theorem proving. An adaptation of Christoph Walther's many-sorted resolution-based calculus [Walther83, Walther84] will be presented as a prelude to the incorporation of these ideas into the tableau framework. In particular we will show how the many-sorted unification algorithm can be adapted for the system HARP.

7.1 Overview of Many-Sorted Logic

In many-sorted logic, the universe of individuals is broken up into a hierarchy of sorts. This taxonomic classification of the entities in the domain of discourse constrains the range of the variables of this language. In the usual one-sorted logical language, the variables range over the entire universe of individuals. The variables of many-sorted logic are only allowed to range over the individuals of their declared sort.

This typing of variables is quite similar to the type declarations found in the Pascal-like programming languages. By typing the variables, a more concise statement of the input problem can be formulated. Variable sort declarations (type declarations which are called *signatures*) do not have to be repeated for each individual formula that utilizes this information. Instead these declarations are explicitly recorded so that different logical sentences may have common access to this information. The sort declarations and the sort hierarchy are formulated and kept in one localized area. Unnecessary duplication of this information is avoided. A problem with a natural many-sorted structure (i.e. the entities of that domain are classified in a taxonomic structure) will benefit from this technique. Clearly, the many-sorted formulation of Schubert's Steamroller (see Figure 7-1) is shorter and more concise than its original one-sorted counterpart presented in section 6.3. A formal description of the language of many-sorted logic will be presented

later in this chapter.

- 1) type $w:W, f:F, b:B, c:C, s:S, g:G, a:A, p:P$
- 2) sort $W, F, B, C, S \ll_S A$
- 3) sort $G \ll_S P$
- 4) $\forall a_1 (\forall p \text{ Eats}(a_1, p)) \vee (\forall a_2 M(a_2, a_1) \wedge (\exists p \text{ Eats}(a_2, p)) \Rightarrow \text{Eats}(a_1, a_2))$
- 5) $\forall c \forall b M(c, b)$
- 6) $\forall s \forall b M(s, b)$
- 7) $\forall b \forall f M(b, f)$
- 8) $\forall f \forall w M(f, w)$
- 9) $\forall f \forall w \neg \text{Eats}(w, f)$
- 10) $\forall g \forall w \neg \text{Eats}(w, g)$
- 11) $\forall b \forall c \text{ Eats}(b, c)$
- 12) $\forall b \forall s \neg \text{Eats}(b, s)$
- 13) $\forall c \exists p \text{ Eats}(c, p)$
- 14) $\forall s \exists p \text{ Eats}(s, p)$
- 15) $\exists a_1, a_2 \forall g \text{ Eats}(a_1, a_2) \wedge \text{Eats}(a_2, g)$

Figure 7-1: Many-sorted axiomatization of Schubert's Steamroller.

The type declarations, statement 1, restrict the range of the variables to that of its assigned sort (in this example, the first letter of the variable denotes its sort; eg. a_1 ranges over the sort of animals). Statements 2 and 3 describe the sort hierarchy (the symbol $\ll_S$ denotes a direct subsort order). Figure 7-2 shows the sort hierarchy of this problem. The many-sorted axiomatization presented in Figure 7-1 is much more concise than the original first-order version described in section 6.3 because of the natural many-sorted structure of this domain.

Essentially, in the translation to a many-sorted logic, the monadic predicates (the predicates with only one argument position) are analyzed to produce the sort hierarchy and declarations. Formulas or subformulas of the form $\forall x (\text{Sort-type-}a(x) \Rightarrow F)$ are reduced to $\forall x_a F'$ where F is any arbitrary wff, x_a is a variable which is declared to be of Sort-type- a , and F' is the same as F except all occurrences of x are replaced by x_a .

The intuitive appeal of this style of logic is the seemingly smaller search space that results from a more concise statement of the input problem. This advantage, however, is only gained when the domain itself is composed of these distinguishable sorts. There are

many problems which do not have this many-sorted universe. For example, consider the problem constructed by replacing each monadic sort predicate in Schubert's Steamroller with a corresponding dyadic predicate having a constant, say d , in the extra argument position.

Figure 7-2: Sort hierarchy of Schubert's Steamroller.

However, as previously alluded, there are also many naturally occurring problems which do have a many-sorted domain. For example, it appears that many of the traditional database applications have a many-sorted structure. The universe of the domain is partitioned into distinct entity sets. For example, a common application is the employee, department, project, and manager database. These entity sets correspond directly to the sorts of many-sorted logic. The set of managers is a subsort of the employees who are themselves a subsort of the set of all people. Logical axioms describing these domains would be able to take advantage of the simplifying translation to a many-sorted formulation.

Computational techniques such as many-sorted unification have been developed

(eg. see [Walther83]) to exploit the information expressed by the sort hierarchy. This information causes a restriction of the unification procedure. The many-sorted proof procedures seem to be more effective because they deal with a more concise formulation of the problem and because the restriction of the unification procedure prunes the space of possible operations.

As a passing remark, it appears that these benefits are also applicable to the construction of rule-based inference engines, particularly those that are integrated with database management systems. Since the entities of the information modelled by databases are often in a natural many-sorted structure (eg. people, employees, departments, projects, managers), it appears that a rule-based architecture adapted from the techniques found in many-sorted logic systems is appropriate for this type of application. The advantages of this unique architecture would be a more natural and concise representation of the knowledge along with an associated increase in the deductive efficiency. For example, suppose we wanted to represent the following piece of knowledge :

IF there is a project to be worked on
 and an employee has previous work experience with a similar project
 THEN that employee is suitable for that project

In the traditional one-sorted rule representation, this knowledge is written in the following format:

$$\begin{aligned} \forall x \forall y \forall z \text{ IF } & \text{Employee}(x) \\ & \wedge \text{Project}(y) \\ & \wedge \text{Project}(z) \\ & \wedge \text{Similar}(y,z) \\ & \wedge \text{Has-previous-experience}(x,y) \end{aligned}$$

THEN Is-suitable-for(x, z)

In a many-sorted format the following shorter representation is employed:

$$\begin{aligned} &\forall e \forall p_1 \forall p_2 \text{ IF } \text{Similar}(p_1, p_2) \\ &\quad \wedge \text{Has-previous-experience}(e, p_1) \\ &\text{ THEN Is-suitable-for}(e, p_2) \end{aligned}$$

7.1.1 Syntax

The language of a many-sorted calculus differs from its one-sorted counterpart because of the division of the universe of individuals into sorts and subsorts. To describe this division a finite and nonempty set of *sort symbols* S is employed. Associated with this set of sort symbols is a partial order $\leq_S$. Let s_1 and s_2 be sort symbols, s_1 is a subsort of s_2 iff $s_1 \leq_S s_2$. Every individual of type sort s_1 is also a member of the sort s_2 . The notation $s_1 <_S s_2$ is an abbreviation for $s_1 \leq_S s_2$ and $s_1 \neq s_2$. We say s_1 is a direct subsort of s_2 , $s_1 <_S s_2$, iff $s_1 <_S s_2$ and there is no $s \in S$ such that $s_1 <_S s <_S s_2$.

The ordered set $\langle S, \leq_S \rangle$ will be restricted to those sets which contain a *maximal element* s_0 where $s \leq_S s_0$ for all $s \in S$. This ordered set is called a *tree structure* when for all $s, s_1, s_2 \in S$, $s \leq_S s_1$ and $s \leq_S s_2$ implies $s_1 \leq_S s_2$ or $s_2 \leq_S s_1$. As we shall see later the shape of this ordered set of sort symbols plays an important role in the efficiency of the proof procedure of this calculus.

We shall now define the predicate, variable, and constant symbols of this many-sorted language. Let S^* denote the set of all finite strings from S , along with the empty string ϵ . For each $s \in S$ and $w \in S^*$, let V_s, C_s, P_w be respectively, pairwise disjoint sets of variable symbols, constant symbols, and predicate symbols. In addition, each sort is guaranteed to be nonempty by the assumption that for each s which is *minimal*

in $\langle S, \leq_S \rangle$ (i.e. there does not exist $s' \in S$ such that $s' <_S s$), C_s is nonempty. The sets V, P, C , and T can now be defined as their first order counterparts in Chapter 2 as respectively, the set of all variable symbols, $\bigcup_{s \in S} V_s$; the set of all predicate symbols, $\bigcup_{w \in S^*} P_w$; the set of all constant symbols, $\bigcup_{s \in S} C_s$; and the set of terms, $T = V \cup C$. An S -sorted signature Σ is the family of all these sets.

Again as in section 2.1, in order to define the many-sorted language of clauses, functional symbols are needed. For each $s \in S$ and $w \in S^*$, $w \neq \epsilon$, $F_{w,s}$ are a set of functional symbols whose rangesort is s . A well-sorted term for the many-sorted language of clauses is either a variable, constant, or a functional expression. A functional expression, say $f(a_1 \dots a_n)$, is well-sorted if it satisfies the following criteria:

- 1) there exists w such that $w' = s_1 \dots s_n$
- 2) $a_i \leq_S s_i$, $1 \leq i \leq n$
- 3) $f \in F_{w',s}$

The rangesort of a term a , denoted by $[a]$, is s iff $a \in V_s \cup C_s \cup F_{w,s}$ (for some $w \in S^*$). For a predicate symbol P , the i th domainsort $[P]_i$ of P is s_i iff $P \in P_w$ where $w = s_1 \dots s_k$, $1 \leq i \leq k$.

Given an S -sorted signature Σ , an atomic formula $P(t_1, \dots, t_n)$ is called a *well-sorted atom* or Σ -atom iff $[t_i] \leq_S [P]_i$, $1 \leq i \leq n$. AT_Σ is the set of all Σ -atoms and LIT_Σ is the set of all Σ -literals. The set of well-sorted formulas are those formulas constructed from atoms of the set AT_Σ . The construction of these well-sorted formulas follow syntax rules similar to those of section 2.1.

A well-sorted clause, Σ -clause, is a finite set of Σ -literals. The many-sorted clausal language L_Σ is the set of all Σ -clauses. $AT_{\Sigma_{gr}}$, $LIT_{\Sigma_{gr}}$, $L_{\Sigma_{gr}}$ are respectively the variable free Σ -atoms, Σ -literals, and Σ -clauses.

The formulas of a many-sorted language can be translated to formulas of first-order.

logic through a process which introduces *sort axioms* and *relativizations*. The sort axioms encode information about the type of the constants and the sort hierarchy. The relativization of a well-sorted formula expresses the sort of each variable by using the sort symbols as unary predicates.

Given an S -sorted signature Σ , the set A^Σ of all sort axioms of Σ is defined to be the smallest set of formulas which satisfy the following properties:

- 1) for each constant symbol a , the atomic formula $s(a) \in A^\Sigma$ if $a \in C_s$.
- 2) if $s_1, s_2 \in S$ and $s_1 \ll_S s_2$, then the formula $\forall x (s_1(x) \Rightarrow s_2(x)) \in A^\Sigma$;
alternatively, in clausal notation, this is expressed as $\{-s_1(x), s_2(x)\} \in A^\Sigma$

For a many-sorted formula F , the relativization F^Σ is achieved by extending F with a precondition restricting the applicability of the formula to only those situations where its variables are instantiated to individuals of the proper sort. For example, suppose e is a variable of type employee and n is a number, consider the many-sorted formula:

$$\forall e \exists n \text{ Salary}(e, n)$$

The relativization of this formula is:

$$\forall x \exists y (\text{Employee}(x) \Rightarrow \text{Number}(y) \wedge \text{Salary}(x, y))$$

The relativization of a Σ -clause is achieved in a similar manner.

7.1.2 Semantics

The same semantic notions described in section 2.2 also apply to this many-sorted language. The primary differences center around the rules of the valuation function $\wp$ which assign truth values to the universally and existentially quantified formulas. The

following rules replace the ones specified in section 2.2:

- 8) $\forall x (A)$: if for every assignment of entities in the declared sort of x to x , $\wp(A) = T$ then $\wp(\forall x (A)) = T$ else F
- 9) $\exists x (A)$: if for at least one assignment of entities in the declared sort of x to x , $\wp(A) = T$ then $\wp(\exists x (A)) = T$ else F

7.2 Many-Sorted Resolution

Recently, Christoph Walther has introduced a many-sorted calculus based on resolution and paramodulation (an inference rule designed to make deductions based on equality) [Walther83]. In this work, a many-sorted unification algorithm was presented (also described in [Walther84]) which, when incorporated into the deductive machinery, resulted in a sound and complete proof procedure. By building the sort information directly into the inference machinery, the size of the search space is constrained and the resulting theorem prover is more efficient for certain types of problems. In addition, results showing the connection between the many-sorted calculus and the traditional one-sorted version are proved.

Much of the material in this section and the notation adopted in this chapter is derived from [Walther83, Walther84]. We will present an overview of this work to highlight certain important features and observations. Since we have so far scoped our interests to focus only on nonequational theories, this discussion will not be concerned with paramodulation.

Walther's many-sorted calculus utilizes the many-sorted clausal language, L_Σ , which differs from the language of first-order logic as outlined in section 7.1. Formal semantics are specified for this many-sorted language whereupon a sound and complete deductive apparatus is presented. It is then shown that the theories of this many-sorted language are in direct correspondence to the theories of first-order logic. Let $\vdash_\Sigma$ and $\models_\Sigma$

denote respectively, derivability and semantic entailment in the many-sorted language.

Figure 7-3 describes these results.

$$\begin{array}{ccc}
 \models_{\Sigma} \phi & \text{iff (1)} & \vdash_{\Sigma} \phi \\
 \\
 \text{iff (2)} & & \text{iff (3)} \\
 \\
 A^{\Sigma} \models \phi^{\Sigma} & \text{iff (4)} & A^{\Sigma} \vdash \phi^{\Sigma}
 \end{array}$$

Figure 7-3: This figure shows the components of Walther's many-sorted calculus and their relationship to FOL. A^{Σ} are the additional first order formulas (the sort axioms) that are constructed in the translation of the many-sorted theorem ϕ to its relativized one-sorted counterpart ϕ^{Σ} .

The proof of conditions 1 and 2 are outlined in [Walther83]. Condition 4 is a standard result achieved by many of the proof procedures of first-order logic (eg. standard resolution). Because of the circularity of this argument, it is not necessary to show condition 3.

We will now outline the deductive apparatus that Walther presents with the many-sorted clausal language. A discussion of many-sorted unification is crucial to this presentation and is the primary contribution of Walther's work. We will cross reference many of the concepts introduced in our earlier discussion of the standard unification algorithm in section 3.2.

A legitimate Σ -substitution σ for an S -sorted signature satisfies the following criterion:

$$[\sigma(x)] \leq_S [x], \text{ for all } x \in \text{DOM}(\sigma)$$

It should be observed that that this definition of a Σ -substitution differs from Walther's definition which is the following:

"A Σ -substitution is a substitution σ satisfying $\sigma(T_\Sigma)$ is a subset of T_Σ ."
[Walther83, pp.15]

T_Σ is the set of terms of the many-sorted language which in our language definition is the union of the variables V and the constants C . The problem with this definition is that it allows substitutions of the form $[\sigma(x)] >_S [x]$, $x \in V$. The inference rule accepting these substitutions could generate deductions which are not sound. Fortunately, this problem is circumvented in the actual implementation as Walther's many-sorted unification procedure does not generate these anomalous substitutions.

Let SUB_Σ denote the set of all Σ -substitutions. A Σ -ground substitution is a Σ -substitution whose codomain is a subset of the set of constants C . The notation $SUB_{\Sigma gr}$ is used to represent the set of all Σ -ground substitutions.

A Σ -substitution μ is a weakening substitution for a set of variables ζ iff it satisfies the following criteria:

- 1) $COD(\mu)$ is a subset of V
- 2) $COD(\mu) \cap \zeta = \emptyset$
- 3) $\mu|_\zeta$, μ restricted to the domain of ζ , is injective
- 4) $[\mu x] <_S [x]$, if $x \in DOM(\mu)$

Let $WSUB(\zeta)$ represent the set of all weakening substitutions for the set of variables ζ .

The deductive apparatus of Walther's calculus is centered around the many-sorted resolution inference rule. This inference rule is essentially the same as its one-sorted counterpart described in section 3.2 except that many-sorted unification or Σ -unification is used to compute the most general unifying Σ -substitutions. The proof procedure, as before, is refutation-based and successfully concludes when the empty clause is derived.

However, as will be discussed later, depending on the shape of the sort hierarchy, the most general Robinson-type unifier (see section 3.2) may be a single unifying substitution or a set of unifying substitutions. In the first case, one resolvent is generated from the application of the resolution inference rule while in the latter case, a set of resolvents are produced, one for each unifying substitution.

The Σ -unification algorithm is obtained by replacing each of the symmetrical statements marked by an * in the unification algorithm presented in section 3.2 by statements of the following form:

let x be some variable, t be a term;

- 1 if $[t] \leq_S [x]$ then return $\{x \leftarrow t\}$ endif
- 2 if $t \notin V$ or $[t] \otimes_S [x] = \emptyset$ then return unsuccessful unification endif
- 3 if $[x] <_S [t]$ then return $\{t \leftarrow x\}$ endif
- 4 let $\{s_1, \dots, s_k\} = \max([t] \otimes_S [x])$
- 5 let $\{z_1, \dots, z_k\}$ be a subset of V such that no z_i has been previously used and $[z_i] = s_i$
- 6 return $\{\{x \leftarrow z_1, t \leftarrow z_1\}, \dots, \{x \leftarrow z_k, t \leftarrow z_k\}\}$

where $s_1 \otimes_S s_2 = \{s \in S \mid s \leq_S s_1 \text{ and } s \leq_S s_2\}$ and
 $\max(\xi) = \{s \in \xi \mid \text{there does not exist } s' \in \xi \text{ such that } s <_S s'\}$

Depending on the shape of the sort hierarchy, the Σ -unification algorithm may return a single unifier or a set of unifiers. To illustrate, consider the sort hierarchy of Figure 7-4a. Now suppose x , a variable of sort A , and y , a variable of sort B , are to be unified. In this case a single unifier will be returned whereas with the sort hierarchy of Figure 7-4b a set of two unifiers will be returned. Clearly, if the sort hierarchy is tree-shaped then either condition 1 or 3 will apply in the event of a successful unification, thus guaranteeing that a single unifier will always be returned in this situation.

Figure 7-4a

Figure 7-4b

We will now conclude this section with a few comments about the limitations of this calculus. First of all, it assumes that each sort is nonempty. This assumption is explicitly stated in the syntactic definition of the language and is necessary to ensure soundness of the proof procedure. To illustrate, Figure 7-5 depicts the undesirable situation resulting from the relaxation of this assumption.

$$\begin{array}{l} \forall x P(x) \Rightarrow R(x) \\ \forall x P(x) \Rightarrow \neg R(x) \\ \hline \neg P(x) \end{array}$$

Figure 7-5a

$$\begin{array}{l} \forall p R(p) \quad p:P \\ \forall p \neg R(p) \\ \hline \text{contradiction} \end{array}$$

Figure 7-5b

Figure 7-5: The premises of 7-5a are satisfiable while those of 7-5b are not.

In Walther's many-sorted calculus, terms are assigned their sorts a priori. This places limitations on the expressive capability of the calculus. The sort declarations constrain the unification procedure and therefore directly affects the behavior of the theorem prover. Sort declaration-like information expressed within the premises through the use of unary characteristic predicates is not accessible to the unification procedure. To illustrate, suppose Tweety is declared to belong to the animal sort A. The unification

procedure has no way of accessing the information that Tweety is also a member of the sort of birds if that information is derived or even expressed as a premise, such as with a characteristic predicate B in the form $B(\text{Tweety})$. It could very well be possible that certain crucial deductions can not be performed because of failed unifications (eg. proving that Tweety can fly from the additional premise $\forall b:B \text{ Flys}(b)$). It appears that a mechanism which monitors inferences to dynamically modify the sort hierarchy and type declarations is needed to overcome this limitation.

In Schubert's Steamroller the sorts, such as the wolves, foxes, and birds, were all disjoint. In such a case the resulting sort hierarchy will be tree-shaped which ensures the Σ -unification algorithm will only produce a single unifier. Sorts may not be disjoint and yet the sort hierarchy could still be tree-shaped. But this would only be true if the overlaps are not explicitly declared in the sort hierarchy during the problem formulation. If the sort hierarchy is not tree-shaped then a performance degradation of the proof procedure may result from the return of several unifiers from the Σ -unification algorithm.

As a final observation, it should be mentioned that after performing a simple modification of the formulas, the conventional unification algorithm can be applied instead of the Σ -unification algorithm when the sort hierarchy is tree-shaped. In this case, each term is replaced by a series of functions which specify its sort and supersorts. Consider the example from [Stickel86] where $\text{Eats}(A(B(x)), P(y))$ is used to represent "birds are animals which like to eat plants".

7.3 Many-Sorted Tableaux

For reasons of trying to construct a natural and humanly plausible theorem prover (see Chapter 1), HARP does not use the unification algorithm. However it is still possible to implement modifications to the matching and instantiation process of HARP

that are similar to the modifications conducted by Walther in the construction of his many-sorted unification algorithm.

The input to HARP would be expressed in the full many-sorted language. The matching process would utilize an explicitly declared sort hierarchy to constrain candidate matches. The instantiation mechanism would only instantiate to symbols automatically flagged to indicate that they belong to the declared sort of the variable.

A many-sorted matching algorithm is created from the match procedure of section 5.1 by changing the condition specified in the if statement marked by the * to a call to the following boolean procedure:

Not-matching-U-variables (arg1,arg2)

if arg1 is a universally quantified variable and $[arg2] \leq_s [arg1]$
then return false endif

if arg2 is a universally quantified variable and $[arg1] \leq_s [arg2]$
then return false endif

if arg1 and arg2 are both universally quantified variables
and $[arg1] \otimes_s [arg2] \neq \emptyset$
then return false endif

return true

End Not-matching-U-variables

If the terms will unify under the Σ -unification algorithm then they will match under this algorithm. Analogous to the original case, the modified match procedure is a relaxed form of the Σ -unification algorithm. These extensions have not been implemented yet for the system HARP.

8 CONCLUSIONS

We have presented in this document an efficient, human-oriented theorem prover as a potential tool in AI environments. HARP attempts to narrow the gap between logic and AI by combining general problem solving methods of logic with easily constructed specialized inference strategies. We believe this work has shown that it is possible to have a uniform, declarative representation which can, under heuristic control, produce efficient specialized behavior. To facilitate experimentation with different sets of heuristics - an ongoing concern of ours - we have separated the basic control structure from explicitly stated heuristics.

By employing the past two decades of AI research developments we have motivated the design of our deductive architecture. Following the inefficient systems of the earlier theorem proving research and the procedural/declarative controversy to the emerging realization that logic is of fundamental importance to AI both as a tool for the analysis of representational schemes and as a basis for new representations, we have developed the system HARP in the spirit of utilizing logic within an unique architecture which synthesizes specialized heuristics tailored behavior along with newly developed general reasoning mechanisms.

The heuristics serve to muffle the criticisms the proceduralists have voiced at logic. They allow deduction to be controlled in an efficient and specialized manner which can be tailored to the specific application domain.

The general reasoning mechanisms can be used separately from any domain dependent heuristics and when actually utilized in this manner, the system reaches a high level of efficiency. This claim is empirically justified by the quick solution achieved for the original first-order version of the challenge problem, Schubert's Steamroller.

The primary objective of this thesis is to narrow the gap in the slippery relationship

between logic and mainstream AI. We have attempted to clarify and bridge the gap between the formal logic-based automated reasoning community and the mainstream AI community. We believe that future work with representation formalisms should be grounded and described within a well-defined and well understood foundation. To this end, a framework for representation and reasoning (designed for an AI environment and based upon logic) has been constructed.

Our framework utilizes heuristics in a manner which silences much of the criticisms that the proceduralists have voiced at logic. Our nontheorem detection mechanism makes plausible the possibility of operation in an AI environment. The entire methodology is supported by and also relies on the use of the full language of first-order logic. In addition, a synergistic relationship between the proof condensation technique, a general purpose efficiency mechanism, and the above heuristic framework has been established.

In addition, insofar as the members of the automated theorem proving community are concerned, HARP represents a new approach to general purpose theorem proving which is more efficient, flexible, and natural than the traditional resolution-based methods.

8.1 Future Work

We will now outline some recommended areas of interest which future work should focus upon.

It appears that tableaux are amenable to a parallel architecture. The processing of each branch can be performed independently of the others. We feel that a study should be conducted to determine the effects of parallelism on HARP's performance. For each branch point, HARP constructs two subtrees that can be developed independently of one another. This fact can be exploited in a parallel architecture.

The detection of nontheorems is crucial to the effective operation of a theorem prover in a general AI environment. To this end, our preliminary work serves as a starting point for future activity in this fundamentally important area. More sophisticated strategies for detecting consistency need to be developed. The architecture of HARP is an excellent vehicle to incrementally formulate and test these heuristics. Devices such as the proof condensation mechanism can serve as metrics for evaluation purposes.

HARP should be extended to handle functional terms in a cognitively plausible fashion. Many problems are often naturally expressed using functions and it becomes difficult and tedious to translate them to a corresponding predicate format. The ability to perform equality-based reasoning is often desirable with problems of a functional nature. By explicitly building this capability directly into the inference machinery, the resulting theorem prover may become more efficient. We feel that a strategy more cognitively plausible than skolemization and unification is desirable.

We believe that a theory of indexing should be formulated to extend the work on non-syntactic schemes that started with the development of associative networks. We feel such a theory is crucial to the development of effective knowledge representation frameworks. In our vision, the indexing scheme of the representation frameworks of the future will be intricately linked to the ongoing resource-limited processing and reasoning mechanisms operating upon the representation expressions. Indexing determines the relevance of pieces of knowledge to the specific context. Indexing determines the overall importance of the pieces of knowledge. Indexing thus determines which pieces of knowledge are remembered and which pieces are forgotten.

More work is needed on integrating procedures within HARP. The heuristics offers a procedural-type advice over a wide range of operations. However it is still possible to extend HARP by building in computable predicates to capture more of the

specialized calculations performed by the procedural demons within frame systems. For example, these computable predicates can calculate default values in a fashion similar to the if-needed demons of the frame systems. In this case, the procedure performs a skolemization-like task but the reduction may possibly lead to previously known constants.

These computable predicates may also perform functions which duplicate the if-added demons which execute when slot values are added. This may be accomplished within HARP by executing the procedure attached to the predicates used in the insertion of additional premises. By attaching procedures to predicates, the potential for these capabilities will be present.

As previously alluded, we believe it is important to incorporate more of the frame-based style of reasoning into a logical framework. This style of processing, of making inferences, of structuring the manipulations to properly interpret incoming sensory information is attractive. Not only is there a structure which can determine the validity of theorems but also something that will indicate which theorems to test (i.e. which theorems are to be used as input). For example, theorems corresponding to the expectations of the current situation are viable candidates.

More work on the logical analysis of this frame-like style of reasoning is needed to clearly understand what role these mechanisms are to play in the AI programs of the future. This style of reasoning appears to require the successful solution of many of the problem areas previously described in this section. Concepts such as indexing, heuristics, and nontheorem detection are prerequisites to this human-like style of reasoning.

8.2 Results and Observations

We will now recapitulate the features of HARP which will summarize the highlights of this thesis.

Analytic tableaux have been used to provide a natural control structure. It appears that the theorem proving community has generally ignored tableaux because of their presumed inefficiency (for a few notable exceptions see [Wrightson84, Wrightson86, Schonfeld85]). On the other hand, many logicians find tableaux to yield an elegant and natural codification of logic (e.g. [Fitting83]). In this thesis we have presented evidence that tableaux, augmented with an indexing scheme to support proof condensation and heuristics, form a basis for combining both efficiency and naturalness.

Since HARP is intended to be used both interactively and as a tool in AI applications, we have attempted to at least partially achieve the design goals stated at the outset:

- 1) utilization of an expressive representation language with well-defined semantics,
- 2) clear understanding of the inferential properties of this language,
- 3) naturalness,
- 4) efficiency,
- 5) indexing to eliminate irrelevant premises and to highlight pertinent ones,
- 6) rapid detection of certain types of nontheorems, and
- 7) partial specifiability of the control structure by heuristic rules.

In the course of trying to realize these goals, we have made the following observations:

- Logical formalisms have well-defined semantics which permit a clear understanding of their inferential properties.
- First-order logic is a very expressive representation language that can be used to analyze alternative representation proposals. Under such analysis,

these formalisms (eg. semantic networks) are discovered to be essentially a subset of FOL.

- Within competing logic-based formalisms, naturalness is achieved by avoiding conversion to any normal form and by working with common inference patterns like Modus Ponens, Universal Instantiation, etc.
- The use of the full, unmodified language of first-order logic eliminates inefficiencies inherent in the language of clauses and makes it possible to attach heuristic advice to premises because extra-logical control information is preserved.
- The inference steps of the tableau calculus facilitate the development of heuristics simulating human reasoning patterns. Some heuristics in HARP correspond to sequences of inference steps that are packaged together as admissible rules. These condensed deductions produce elegant and short proofs.
- Ordinary logical notation and common inference patterns aid in interactive theorem proving by making the intermediate proof steps more intelligible than they would be in a system based on clauses and resolution.
- An architecture modifiable by declarative heuristics is desirable in automated theorem proving. The iterative development cycles employed successfully in the creation of knowledge-based systems can be applied to assist in the

enhancement of HARP. An extensible architecture has the further advantage of making it straightforward to add specialized control regimes to perform certain inference types (e.g. inheritance) efficiently.

- Most of the heuristics are nonlocal, i.e. they are formulated in terms of properties of entire branches of proof trees. They encode control strategies to construct proofs, strategies to detect nontheorems, and domain specific knowledge to facilitate specialized inferences. Other heuristics make commonsense knowledge available by supplying missing background information.

- Heuristics for determining potentially infinite branches, i.e. for detecting nontheorems, are essential to any theorem prover operating in an AI environment. Such heuristics have to be based on global patterns of branches in proof trees.

- Proof condensation is an effective procedure for eliminating redundancies. By shortening proofs it contributes both to greater efficiency and to greater clarity. Its frequency of operation indicates shortcomings in the current set of heuristics, making it a helpful tool in the gradual development of the heuristics.

- The indexing scheme helps in weeding out irrelevant premises. Some of its uses constitute a first step towards the goal of selecting not just syntactically relevant but also semantically relevant premises. This feature along with the

admissible inference rules captures the intended behavior of structured knowledge representation formalisms such as frames (see also [Hayes79]).

- HARP's relatively good performance as a consistency checker - due to the heuristics that determine when an open branch is to be declared complete - makes it suitable for applications such as truth maintenance and planning.

REFERENCES

- [Allen84] Allen, J.F. "Towards a General Theory of Action and Time", *Artificial Intelligence*, Vol.23, No.2, 1984, pp. 123-154.
- [Andrews81] Andrews, P.B. "Theorem Proving via General Matings", *Journal of the ACM* 28, 2, pp.193-214, 1981.
- [Barr81] Barr, A. and Feigenbaum, E.A. (eds), *The Handbook of Artificial Intelligence*, Volume 1, William Kaufmann Publishers, Inc. 1981.
- [Bibel79] Bibel, W. "Syntax Directed, Semantic Supported Program Synthesis", *4th International Conference on Automated Deduction*, 1979, pp.140-147.
- [Bibel82] Bibel, W. *Automated Theorem Proving*, Vieweg, 1982.
- [Bibel83] Bibel, W. "Matings in Matrices", *Communications of the ACM*, Vol. 26, No. 11, pp.844-852, 1983.
- [Biermann84] Biermann, A.W., Guiho, G., and Kodratoff, Y. *Automatic Program Construction Techniques*, Macmillan Publishing Company, 1984, pp 570
- [Bledsoe83] Bledsoe, W.W., "The UT Interactive Prover", The University of Texas at Austin, ATP-17B, 1983.
- [Bobrow75] Bobrow, D.G. and Collins, A.M. (eds), *Representation and Understanding: Studies in Cognitive Science*, New York: Academic Press, 1975.
- [Bobrow77] Bobrow, D.G. and Winograd, T. "An Overview of KRL, A Knowledge Representation Language", *Cognitive Science*, Vol. 1, No. 1, 1977, pp. 3 - 46.
- [Boyer79] Boyer, R. and Moore, J. *A Computational Logic*. Academic Press, 1979.
- [Brachman83] Brachman, R.J. "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", *IEEE Computer*, Vol. 16, No. 10, 1983, pp.30-36.
- [Brachman85a] Brachman, R.J. and Levesque, H.J. (eds), *Readings in Knowledge Representation*, Morgan Kaufmann Publishers, Inc. 1985.
- [Brachman85b] Brachman, R.J. and Schmolze, J. "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science*, Vol. 9, No. 2, 1985, pp. 171-216.
- [Buchanan84] Buchanan, B.G. and Shortliffe, E.H. (eds), *Rule-Based Expert Systems*, Addison Wesley, 1984.

-
- [Charniak80] Charniak, E., Riesbeck, C., and McDermott, D. *Artificial Intelligence Programming*, Erlbaum Associates, Hillsdale, N.J., 1980.
- [Charniak85] Charniak, E. and McDermott, D. *Artificial Intelligence*, Addison Wesley Publishing Company, 1985, pp. 701.
- [Cohen81] Cohen, D.N. *Knowledge Based Theorem Proving and Learning*. UMI Research Press, Ann Arbor, Michigan, 1981.
- [Cohn85] Cohn, A.G. "On the solution of Schubert's Steamroller in Many Sorted Logic", *Proc. of the IJCAI-85*, pp. 1169-1173.
- [Davis82] Davis, R., Teiresias: Applications of Meta-Level Knowledge, in Davis, R., and Lenat, D.B. (eds.), *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, Inc., 1982, pp. 229-490.
- [Doyle80] Doyle, J. "A Model for Deliberation, Action, and Introspection", TR-581, MIT Artificial Intelligence Laboratory, 1980.
- [Fikes85] Fikes, R. and Kehler, T., "The Role of Frame-based Representation in Reasoning", *Communications of the ACM*, Vol. 28, No. 9, Sept., 1985, pp.904 - 920.
- [Findler79] Findler, N.V., (ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, 1979.
- [Fitting83] Fitting, M.C., *Proof Methods for Modal and Intuitionistic Logics*, Reidel, Dordrecht, Holland, 1983.
- [Green69] Green, C. "Theorem Proving by Resolution as a Basis for Question Answering Systems", in Meltzer, B. and Michie, D. (eds) *Machine Intelligence 4*, Edinburgh University Press, 1969, pp. 183-208.
- [Halpern86] Halpern, J.Y., *Proc. of the 1986 Conference on the Theoretical Aspects of Reasoning About Knowledge*, California, Morgan Kaufmann Publishers, Inc., 1986.
- [Hayes74] Hayes, P.J., "Some Problems and Non-Problems in Representation Theory", *Proc. AISB Summer Conference*, University of Sussex, 1974, pp.63-79; also in [Brachman85].
- [Hayes77] Hayes, P.J. "In Defense of Logic" in *IJCAI-77*, 1977, pp. 559-565.
- [Hayes79] Hayes, P.J. "The logic of frames" in *Frame Conceptions and Text Understanding* (ed) Metzger, D. Berlin: Walter de Gruyter and Co. 1979, pp. 46-61
- [Hayes-Roth83] Hayes-Roth, F., Waterman, D., and Lenat, D. (eds.) *Building*

- Expert Systems*, Addison-Wesley, 1983.
- [Hendrix79] Hendrix, G.G., "Encoding Knowledge in Partitioned Networks", in [Findler79], pp. 51-92.
- [Hewitt71] Hewitt, C. "PLANNER: A Language for Proving Theorems in Robots", *IJCAI-71*, 1971, pp. 295-301
- [Hewitt75] Hewitt, C. "How to Use What You Know", *IJCAI-75*, 1975, pp. 189-198
- [Hewitt85] Hewitt, C. "The Challenge of Open Systems", *BYTE*, Vol. 10, No. 4, 1985, pp. 223-242.
- [Hunter71] Hunter, G. *Metalogic: An Introduction to the Metatheory of Standard First-Order Logic*, MacMillan and Co LTD, 1971.
- [Jeffrey67] Jeffrey, R.C., *Formal Logic: Its Scope and Limits*, McGraw-Hill, New York, 1967.
- [Kalish64] Kalish, D., Montague, R., *Logic: Techniques of Formal Reasoning*, Harcourt, Brace & World, New York, 1964.
- [Konolige84] Konolige, K., "A Deductive Model of Belief and its Logics", Technical Note 326, SRI International, Menlo Park, CA, 1984
- [Kowalski75] Kowalski, R., "A Proof Procedure Using Connection Graphs", *Journal of the ACM*, Vol. 22, No. 4, Oct. 1975, pp. 572-595.
- [Kowalski79] Kowalski, R.A., *Logic for Problem Solving*, Elsevier North-Holland, Amsterdam (1979).
- [Levesque84a] Levesque, H.J. "A Fundamental Trade-off in Knowledge Representation and Reasoning" in *CSCSI-84*, pp. 141-152, London, Ontario, May, 1984; also in [Brachman85], pp. 41-70.
- [Levesque84b] Levesque, H.J., Foundations of a Functional Approach to Knowledge Representation, *Artificial Intelligence*, Vol. 23, No. 2, 1984, pp. 155-212.
- [Loveland78] Loveland D.W. *Automated theorem proving: a logical basis*, North Holland Publishing Company, pp 405, 1978
- [McCarthy80] McCarthy, J. "Circumscription - A Form of Non-Monotonic Reasoning", *Artificial Intelligence*, Vol. 13 (1,2), 1980, pp.27-39
- [McCarthy69] McCarthy, J. and Hayes, P.J. "Some Philosophical Problems from the Standpoint of Artificial Intelligence" in *Machine Intelligence 4* (eds) Meltzer, B. and Michie, D. Edinburgh University Press, Edinburgh, Scotland, 1969

- [McDermott80] McDermott, D. and Doyle, J. "Non-Montonic Logic I", *Artificial Intelligence*, Vol. 13, (1,2), 1980, pp. 41-72.
- [McDermott82] McDermott, D. "A Temporal Logic for Reasoning about Processes and Plans", *Cognitive Science*, Vol. 6, No. 2, 1982, pp. 101-155.
- [Mendelson64] Mendelson, E. *Introduction to Mathematical Logic*, Van Nostrand, Princeton, 1964, pp. 300
- [Minsky75] Minsky, M. "A Framework for Representing Knowledge" in *The Psychology of Computer Vision*. New York; McGraw-Hill Book Company (ed) Winston, P.H. 1975; also in [Brachman and Levesque 85], pp. 245-262.
- [Moore82] Moore, R.C., "The Role of Logic in Knowledge Representation and Commonsense Reasoning." in *AAAI-82*, 1982, pp. 428-433; also in [Brachman and Levesque 85], pp. 335-341.
- [Murray82] Murray, N.V. "Completely Non-Clausal Theorem Proving", *Artificial Intelligence*, Vol. 18, 1982, pp.67-85.
- [Newell82] Newell, A. "The Knowledge Level", *Artificial Intelligence*, Vol. 18 (1), 1982, pp.87-127.
- [Nilsson80] Nilsson, N. *Principles of artificial intelligence*, Tioga Publishing, pp 476, 1980
- [Oppacher86a] Oppacher, F. and Suen, E. "An Efficient Tableau-Based Theorem Prover", *Proc. of The 6th Canadian Conference on Artificial Intelligence*, 1986.
- [Oppacher86b] Oppacher, F. and Suen, E. "Controlling Deduction with Proof Condensation and Heuristics", *Proc. of The 8th International Conference on Automated Deduction*, 1986.
- [Oppacher87] Oppacher, F. and Suen, E. "HARP: A Tableau-Based Theorem Prover", to appear in the *Journal of Automated Reasoning*
- [Patel-Schneider86] Patel-Schneider, P.F. "A Hybrid, Decidable, Logic-Based Knowledge Representation System", *Proc. of the Sixth Canadian Conference on Artificial Intelligence*, 1986, pp. 210-214.
- [Pelletier82] Pelletier, J.F. "Completely Non-Clausal, Completely Heuristically Driven Automatic Theorem Proving", Technical Report TR82-7, Dept. of Computer Science, Univ. of Alberta, 1982, pp.285
- [Quillian67] Quillian, M.R. "Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities", *Behavioral Science*, Vol. 12, 1967, pp. 410-430; also in [Brachman and Levesque 85], pp. 97-118.
- [Quine72] Quine, W.V., *Methods of Logic*, Holt, Rinehart, and Winston, New

- York, 1972.
- [Reinfrank85] Reinfrank, M. "An Introduction to Non-Monotonic Reasoning", MEMO-SEKI-85-02, Universitat Kaiserslautern, 1985
- [Reiter80] Reiter, R. "A Logic for Default Reasoning", *Artificial Intelligence*, Vol. 13 (1,2), 1980, pp. 81 - 132.
- [Reiter86] Reiter, R. "A Theory of Diagnosis from First Principles", Technical Report No. 87/86, Univ. of Toronto, 1986; also to appear in *Artificial Intelligence*.
- [Rich83] Rich, E., *Artificial Intelligence*, McGraw-Hill, Inc., 1983, pp.436.
- [Robinson65] Robinson, J.A. "A machine-oriented logic based on the resolution principle", *Journal of the ACM*, Vol. 12, No. 1, pp. 23, 1965.
- [Robinson81] Robinson, J.A., *Logic: Form and Function*, North Holland, New York, 1981.
- [Santoro86] Santoro, N., and Suen, E. "Reduction Techniques for Selection in Distributed Files", *Proc. of The 15th Annual International Conference on Parallel Processing*, IEEE, 1986.
- [Schank77] Schank, R.C., and Abelson, R. *Scripts, Plans, Goals, and Understanding*, Hillsdale, NJ: Lawrence Erlbaum Associates, 1977.
- [Schonfeld83] Schonfeld, W. "Proof Search for Unprovable Formulas", *German Workshop in Artificial Intelligence*, 1983, pp.207-215.
- [Schonfeld85] Schönfeld, W., Prolog extensions based on Tableau Calculus, *Proc. of the IJCAI 85*, 1985, pp. 730-732.
- [Schubert76] Schubert, L.K., "Extending the Expressive Power of Semantic Networks", *Artificial Intelligence*, Vol. 7, No. 2, 1976, pp.163 - 198.
- [Smith82] Smith, B.C., Prologue to Reflection and Semantics in a Procedural Language, Ph.D. dissertation and Tech. report MIT/LCS/TR-272, M.I.T., 1982; also in [Brachman85], pp.31 - 40.
- [Smullyan68] Smullyan, R.M., *First-Order Logic*, Springer Verlag, Berlin , 1968.
- [Sowa84] Sowa, J.F., *Conceptual Structures*, Addison Wesley, 1984.
- [Stefik79] Stefik, M., An Examination of a Frame-Structured Representation System, *Proc. IJCAI-79*, 1979, pp. 845-852.
- [Stickel82] Stickel, M.E., A Nonclausal Connection-Graph Resolution Theorem Proving Program, *Proceedings of the AAAI-82 National Conference on Artificial Intelligence*, 1982, pp. 229-233.

-
- [Stickel86] Stickel, M.E., "Schubert's Steamroller Problem: Formulations and Solutions", *Journal of Automated Reasoning*, 2, 1986, pp. 89-101.
- [Turner84] Turner, R. *Logics for Artificial Intelligence*, Ellis Horwood Limited, 1984, pp. 121.
- [Walther83] Walther, C. "A Many-Sorted Calculus Based on Resolution and Paramodulation", *Proc. of the IJCAI-83*, Vol. 2, 1983, pp. 882-891; also Interner Bericht 34/82, Universität Karlsruhe, 1982.
- [Walther84] Walther, C. "Unification in Many-Sorted Theories", *Proc. of the Sixth European Conference on Artificial Intelligence*, North-Holland, 1984; also Interner Bericht 10/85, Universität Karlsruhe, 1985.
- [Walther85] Walther, C. "A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution", *Artificial Intelligence*, May, 1985, pp.217-224; also Interner Bericht 5/84, Universität Karlsruhe, 1984.
- [Winograd75] Winograd, T. "Frame Representations and the Declarative/Procedural Controversy" in [Bobrow75], pp 185 - 210; also in [Brachman85], pp. 357 - 370.
- [Winston81] Winston, P.H. and Horn, B.K.P., *LISP*, Addison-Wesley, 1981.
- [Woods75] Woods, W.A. "What's In a Link: Foundation for Semantic Networks" in [Bobrow75], pp. 35-82; also in [Brachman85], pp. 217-241.
- [Wos84] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., *Automated Reasoning: Introduction and Applications*, Prentice-Hall, Inc., New Jersey, 1984, pp.482.
- [Wrightson84] Wrightson, G., Semantic Tableaux, Unification, and Links, Technical Report CSD-ANZARP-84-001, Victoria University, New Zealand, 1984.
- [Wrightson85] Wrightson, G., Non-Classical Logic Theorem Proving, in L.Wos et.al. An Overview of Automated Reasoning, *Journal of Automated Reasoning*, vol.1, pp.5-48, 1985.

APPENDICES

APPENDICES

These appendices contain the proof trees of a selected sample of problems which have been solved by HARP. The proof trees are displayed in a sideways manner where the branching level of a node of the tree is indicated by the number of tab indentations. Each branch point of the tree is specially marked with the tree level number. As described in the thesis, all nodes are produced from the decomposition of an ancestral node of the proof tree (unless it is a premise or the conclusion). The printed description of each node is followed by the branch level of its ancestral node.

Appendix A-1 contains the proof tree of a theorem which W. Bledsoe's UT theorem prover [Bledsoe83] cannot prove. HARP solves the problem instantaneously. Notice the proof tree is compact; this is a direct result of the proof condensation procedure which eliminates all redundant information to make the proof more natural and readable.

Appendix A-2 contains the formulation of the monkey and banana problem found in [Loveland78]. Again, HARP finds the solution immediately.

Version *** of Schubert's Steamroller is in Appendix A-3, while version ** is in Appendix A-4.

Appendix A-5 is an example of the nontheorem detection capabilities of HARP. A modified version of Schubert's Steamroller problem with missing premises is presented. After some processing, HARP detects an open branch and ceases computation.

Appendix A-1

... Bledsoe's UT theorem prover cannot solve this problem
... from [Bledsoe83]

(prove '(((U z ((Q (z)) -> (P (z)))))
'(E x (((P (x)) -> (P (a))) and ((Q (x)) -> (P (b)))))))

PROVED

(pp-tree2)

Dump of proof --> Number of closing-nodes = 5

(NOT (E X (((P (X)) -> (P (A))) AND ((Q (X)) -> (P (B)))))) from CONCLUSION
(U X (NOT ((P (X)) -> (P (A))) AND ((Q (X)) -> (P (B)))) from area #0
(NOT (E X (((P (X)) -> (P (A))) AND ((Q (X)) -> (P (B)))))) from CONCLUSION
#1 (NOT ((P (B)) -> (P (A))) AND ((Q (B)) -> (P (B)))) from area #0
 (NOT ((P (B)) -> (P (A)))) from br pt #1
 (NOT (P (A))) from area #1
 (U Z ((Q (Z)) -> (P (Z)))) from PREMISES
 (U X (NOT ((P (X)) -> (P (A))) AND ((Q (X)) -> (P (B)))) from area #0
#2 ((Q (A)) -> (P (A))) from area #1
 (NOT (Q (A))) from br pt #2
#3 (NOT (((P (A)) -> (P (A))) AND ((Q (A)) -> (P (B)))) from area #1
 (NOT ((P (A)) -> (P (A)))) from br pt #3
 (P (A)) from area #3
 Branch Closed -> Going back to next branch

 (NOT ((Q (A)) -> (P (B)))) from br pt #3
 (Q (A)) from area #3
 Branch Closed -> Going back to next branch

 (P (A)) from br pt #2
 Branch Closed -> Going back to next branch

 (NOT ((Q (B)) -> (P (B)))) from br pt #1
 (Q (B)) from area #1
 (U Z ((Q (Z)) -> (P (Z)))) from PREMISES
 (NOT (Q (B))) from area #1
#2 ((Q (B)) -> (P (B))) from area #1
 (NOT (Q (B))) from br pt #2
 Branch Closed -> Going back to next branch

 (P (B)) from br pt #2
 Branch Closed -> Going back to next branch

-----End of Tree-----

Appendix A-2

... the monkey and banana problem from [Loveland78]

```
(prove '(U x (U y (((a(x)) and (cl(x y))) -> (r(x y)))))
  (U x (U y (((o(x y)) and (under(y b))) and (t(y))) -> (cl(x b)))))
  (U x (U y (U z (((i(x)) and (i(y)) and (i(z)) and (mv(x y z))) -> ((cl(z f)) or (under(y z))))))
  (U x (U y ((cb(x y)) -> (o(x y)))))
  (a(m))
  (t(c))
  (i(m))
  (i(b))
  (i(c))
  (mv(m c b))
  (not (cl(b f)))
  (cb(m c)))
  (r(m b)))
```

PROVED

(pp-tree2)

Dump of proof --> Number of closing-nodes = 12

```
(A (M)) from NEW-PREMISE
(U X (U Y (((A (X)) AND (CL (X Y))) -> (R (X Y))))) from PREMISES
(U Y (((A (M)) AND (CL (M Y))) -> (R (M Y)))) from area #0
(I (M)) from NEW-PREMISE
(U X (U Y (U Z (((I (X)) AND (I (Y)) AND (I (Z)) AND (MV (X Y Z))) -> ((CL (Z F)) OR (UNDER (Y Z))))))
from PREMISES
(U Y (U Z (((I (M)) AND (I (Y)) AND (I (Z)) AND (MV (M Y Z))) -> ((CL (Z F)) OR (UNDER (Y Z)))))) from
area #0.
(U Z (((I (M)) AND (I (C)) AND (I (Z)) AND (MV (M C Z))) -> ((CL (Z F)) OR (UNDER (C Z)))) from area #0
(I (C)) from NEW-PREMISE
(I (B)) from NEW-PREMISE
(MV (M C B)) from NEW-PREMISE
(NOT (CL (B F))) from NEW-PREMISE
(CB (M C)) from NEW-PREMISE
(T (C)) from NEW-PREMISE
(NOT (R (M B))) from CONCLUSION
#1 (((A (M)) AND (CL (M B))) -> (R (M B))) from area #0
  (U X (U Y (((O (X Y)) AND (UNDER (Y B))) AND (T (Y))) -> (CL (X B))))) from PREMISES
#2 (NOT ((A (M)) AND (CL (M B)))) from br pt #1
  (NOT (A (M))) from br pt #2.
  Branch Closed -> Going back to next branch
  (U Y (((O (M Y)) AND (UNDER (Y B))) AND (T (Y))) -> (CL (M B))) from area #1
  (NOT (CL (M B))) from br pt #2.
#3 (((I (M)) AND (I (C)) AND (I (B)) AND (MV (M C B))) -> ((CL (B F)) OR (UNDER (C B)))) from area #0
#4 (NOT (((I (M)) AND (I (C)) AND (I (B)) AND (MV (M C B)))) from br pt #3
#5 (NOT ((I (M)) AND (I (C)) AND (I (B)))) from br pt #4
#6 (NOT ((I (M)) AND (I (C)))) from br pt #5
  (NOT (I (M))) from br pt #6.
  Branch Closed -> Going back to next branch
  (NOT (I (C))) from br pt #6
  Branch Closed -> Going back to next branch
  (NOT (I (B))) from br pt #5.
  Branch Closed -> Going back to next branch
  (NOT (MV (M C B))) from br pt #4.
  Branch Closed -> Going back to next branch
#4 (((O (M C)) AND (UNDER (C B))) AND (T (C))) -> (CL (M B))) from area #2
#5 (NOT (((O (M C)) AND (UNDER (C B))) AND (T (C)))) from br pt #4
#6 ((CL (B F)) OR (UNDER (C B))) from br pt #3
  (CL (B F)) from br pt #6.
  Branch Closed -> Going back to next branch
```

```

(U X (U Y ((CB (X Y)) -> (O (X Y))))) from PREMISES
(UNDER (C B)) from br pt #6
(U Y ((CB (M Y)) -> (O (M Y))))) from area #6
#7 (NOT ((O (M C)) AND (UNDER (C B))))) from br pt #6
    (NOT (O (M C))) from br pt #7
#8 ((CB (M C)) -> (O (M C))) from area #6
    (NOT (CB (M C))) from br pt #8
    Branch Closed -> Going back to next branch

    (O (M C)) from br pt #8
    Branch Closed -> Going back to next branch

    (NOT (UNDER (C B))) from br pt #7
    Branch Closed -> Going back to next branch

    (NOT (T (C))) from br pt #6
    Branch Closed -> Going back to next branch

    (CL (M B)) from br pt #4
    Branch Closed -> Going back to next branch

    (R (M B)) from br pt #1
    Branch Closed -> Going back to next branch

```

-----End of Tree-----

Appendix A-3

... Schubert's Steamroller problem, version ***

```
(prove '(
  (U x ((W(x)) -> (A(x))))
  (U x ((F(x)) -> (A(x))))
  (U x ((B(x)) -> (A(x))))
  (U x ((C(x)) -> (A(x))))
  (U x ((S(x)) -> (A(x))))
  (E x (W(x)))
  (E x (F(x)))
  (E x (B(x)))
  (E x (C(x)))
  (E x (S(x)))
  (E x (G(x)))
  (U x ((G(x)) -> (P(x))))
  (U x ((A(x)) -> ((U y ((P(y)) -> (Eats(x y)))))
    or
    (U y (((A(y))
      and
      (M(y x))
      and
      (E z ((Eats(y z)) and (P(z)))))
      -> (Eats(x y)))))))
  (U x (U y (((C(x)) and (B(y)))
    -> (M(x y)))))
  (U x (U y (((S(x)) and (B(y)))
    -> (M(x y)))))
  (U x (U y (((B(x)) and (F(y)))
    -> (M(x y)))))
  (U x (U y (((F(x)) and (W(y)))
    -> (M(x y)))))
  (U x ((W(x)) -> ((U y ((F(y)) -> (not (Eats(x y)))))
    and
    (U y ((G(y)) -> (not (Eats(x y)))))
    -> (not (Eats(x y)))))
  (U x ((B(x)) -> ((U y ((C(y)) -> (Eats(x y)))
    and
    (U y ((S(y)) -> (not (Eats(x y)))))
    -> (not (Eats(x y)))))
  (U x ((C(x)) -> (E y ((P(y)) and (Eats(x y)))))
  (U x ((S(x)) -> (E y ((P(y)) and (Eats(x y)))))
  '(E x (E y (((A(x))
    and
    (A(y))
    and
    (Eats(x y))
    and
    (U z ((G(z)) -> (Eats(x z)))))
    -> (Eats(x y))))))
```

PROVED

(pp-tree2)

Dump of proof --> Number of closing-nodes = 49

(NOT (E X (E Y (((A (X)) AND (A (Y))) AND (EATS (X Y)) AND (U Z ((G (Z)) -> (EATS (Y Z))))))) from
CONCLUSION

(E X (W (X))) from PREMISES

(W (C2)) from area #0.

(U X ((W (X)) -> ((U Y ((F (Y)) -> (NOT (EATS (X Y)))) AND (U Y ((G (Y)) -> (NOT (EATS (X Y))))))) from
PREMISES

(E X (B (X))) from PREMISES

(B (C4)) from area #0.

(U X ((B (X)) -> ((U Y ((C (Y)) -> (EATS (X Y))) AND (U Y ((S (Y)) -> (NOT (EATS (X Y))))))) from PREMISES

(E X (C (X))) from PREMISES

(C (C5)) from area #0.

(E X (S (X))) from PREMISES

(S (C6)) from area #0.

(E X (G (X))) from PREMISES

(G (C7)) from area #0.
 (E X (F (X))) from PREMISES
 (F (C3)) from area #0
 #1 ((W (C2)) -> ((U Y ((F (Y)) -> (NOT (EATS (C2 Y)))) AND (U Y ((G (Y)) -> (NOT (EATS (C2 Y)))))) from area #0
 (NOT (W (C2))) from br pt #1
 Branch Closed -> Going back to next branch

 ((U Y ((F (Y)) -> (NOT (EATS (C2 Y)))) AND (U Y ((G (Y)) -> (NOT (EATS (C2 Y)))))) from br pt #1
 (U Y ((G (Y)) -> (NOT (EATS (C2 Y)))) from area #1
 #2 ((B (C4)) -> ((U Y ((C (Y)) -> (EATS (C4 Y)))) AND (U Y ((S (Y)) -> (NOT (EATS (C4 Y)))))) from area #0
 (NOT (B (C4))) from br pt #2
 Branch Closed -> Going back to next branch

 ((U Y ((C (Y)) -> (EATS (C4 Y)))) AND (U Y ((S (Y)) -> (NOT (EATS (C4 Y)))))) from br pt #2
 (U Y ((S (Y)) -> (NOT (EATS (C4 Y)))) from area #2.
 (U Y ((F (Y)) -> (NOT (EATS (C2 Y)))) from area #1
 (U X ((G (X)) -> (P (X)))) from PREMISES
 (U X ((C (X)) -> (E Y ((P (Y)) AND (EATS (X Y)))))) from PREMISES
 #3 ((C (C5)) -> (E Y ((P (Y)) AND (EATS (C5 Y)))) from area #2
 (NOT (C (C5))) from br pt #3
 Branch Closed -> Going back to next branch

 (E Y ((P (Y)) AND (EATS (C5 Y)))) from br pt #3
 ((P (C8)) AND (EATS (C5 C8))) from area #3
 (P (C8)) from area #3.
 #4 ((S (C6)) -> (NOT (EATS (C4 C6)))) from area #2
 (NOT (S (C6))) from br pt #4
 Branch Closed -> Going back to next branch

 (U X ((S (X)) -> (E Y ((P (Y)) AND (EATS (X Y)))))) from PREMISES
 (NOT (EATS (C4 C6))) from br pt #4.
 #5 ((S (C6)) -> (E Y ((P (Y)) AND (EATS (C6 Y)))) from area #4
 (NOT (S (C6))) from br pt #5
 Branch Closed -> Going back to next branch

 (E Y ((P (Y)) AND (EATS (C6 Y)))) from br pt #5.
 ((P (C9)) AND (EATS (C6 C9))) from area #5
 (EATS (C6 C9)) from area #5
 (P (C9)) from area #5
 (U X ((A (X)) -> ((U Y ((P (Y)) -> (EATS (X Y))) OR (U Y (((A (Y)) AND (M (Y X))) AND
 (E Z ((EATS (Y Z)) AND (P (Z)))) -> (EATS (X Y)))))) from PREMISES
 (U X ((B (X)) -> (A (X)))) from PREMISES
 #6 ((A (C4)) -> ((U Y ((P (Y)) -> (EATS (C4 Y))) OR (U Y (((A (Y)) AND (M (Y C4))) AND
 (E Z ((EATS (Y Z)) AND (P (Z)))) -> (EATS (C4 Y)))))) from area #5
 (NOT (A (C4))) from br pt #6
 #7 ((B (C4)) -> (A (C4))) from area #5
 (NOT (B (C4))) from br pt #7.
 Branch Closed -> Going back to next branch

 (A (C4)) from br pt #7.
 Branch Closed -> Going back to next branch

 #7 ((U Y ((P (Y)) -> (EATS (C4 Y))) OR (U Y (((A (Y)) AND (M (Y C4))) AND (E Z ((EATS (Y Z))
 AND (P (Z)))) -> (EATS (C4 Y)))) from br pt #6.
 (U Y ((P (Y)) -> (EATS (C4 Y)))) from br pt #7
 #8 ((P (C8)) -> (EATS (C4 C8))) from area #7
 (NOT (P (C8))) from br pt #8.
 Branch Closed -> Going back to next branch

 (EATS (C4 C8)) from br pt #8.
 #9 ((G (C7)) -> (P (C7))) from area #2.
 (NOT (G (C7))) from br pt #9.
 Branch Closed -> Going back to next branch

 (P (C7)) from br pt #9.

#10 ((G (C7)) -> (NOT (EATS (C2 C7)))) from area #1
 (NOT (G (C7))) from br pt #10
 Branch Closed -> Going back to next branch

(NOT (EATS (C2 C7))) from br pt #10

#11 ((F (C3)) -> (NOT (EATS (C2 C3)))) from area #2
 (NOT (F (C3))) from br pt #11
 Branch Closed -> Going back to next branch

(NOT (EATS (C2 C3))) from br pt #11.
 (U X ((A (X)) -> ((U Y ((P (Y)) -> (EATS (X Y)))) OR (U Y (((A (Y)) AND
 (M (Y X))) AND (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (X Y)))) from PREMISES
 (U X ((W (X)) -> (A (X)))) from PREMISES

#12 ((A (C2)) -> ((U Y ((P (Y)) -> (EATS (C2 Y)))) OR (U Y (((A (Y)) AND
 (M (Y C2))) AND (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (C2 Y)))) from area #11
 (NOT (A (C2))) from br pt #12

#13 ((W (C2)) -> (A (C2))) from area #11
 (NOT (W (C2))) from br pt #13
 Branch Closed -> Going back to next branch

(A (C2)) from br pt #13
 Branch Closed -> Going back to next branch

#13 ((U Y ((P (Y)) -> (EATS (C2 Y)))) OR (U Y (((A (Y)) AND (M (Y C2))) AND
 (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (C2 Y)))) from br pt #12
 (U Y ((P (Y)) -> (EATS (C2 Y)))) from br pt #13

#14 ((P (C7)) -> (EATS (C2 C7))) from area #13
 (NOT (P (C7))) from br pt #14
 Branch Closed -> Going back to next branch

(EATS (C2 C7)) from br pt #14.
 Branch Closed -> Going back to next branch

(U Y (((A (Y)) AND (M (Y C2))) AND (E Z ((EATS (Y Z)) AND (P (Z))))))
 -> (EATS (C2 Y)))) from br pt #13

#14 (((A (C3)) AND (M (C3 C2))) AND (E Z ((EATS (C3 Z)) AND (P (Z)))))) ->
 (EATS (C2 C3))) from area #13

#15 (NOT (((A (C3)) AND (M (C3 C2))) AND (E Z ((EATS (C3 Z)) AND
 (P (Z)))))) from br pt #14

(U X ((F (X)) -> (A (X)))) from PREMISES

#16 (NOT ((A (C3)) AND (M (C3 C2)))) from br pt #16
 (NOT (A (C3))) from br pt #16

#17 ((F (C3)) -> (A (C3))) from area #15
 (NOT (F (C3))) from br pt #17.
 Branch Closed -> Going back to next branch

(A (C3)) from br pt #17.
 Branch Closed -> Going back to next branch

(U X (U Y (((F (X)) AND (W (Y))) -> (M (X Y)))) from PREMISES
 (U Y (((F (C3)) AND (W (Y))) -> (M (C3 Y)))) from area #16
 (NOT (M (C3 C2))) from br pt #16

#17 (((F (C3)) AND (W (C2))) -> (M (C3 C2))) from area #16

#18 (NOT ((F (C3)) AND (W (C2)))) from br pt #17
 (NOT (F (C3))) from br pt #18.
 Branch Closed -> Going back to next branch

(NOT (W (C2))) from br pt #18.
 Branch Closed -> Going back to next branch

(M (C3 C2)) from br pt #17.
 Branch Closed -> Going back to next branch

(NOT (E Z ((EATS (C3 Z)) AND (P (Z)))))) from br pt #16
 (U Z (NOT ((EATS (C3 Z)) AND (P (Z)))))) from area #15.

(U X (U Y ((B (X)) AND (F (Y))) -> (M (X Y)))) from PREMISES
 #16 (NOT ((EATS (C3 C7)) AND (P (C7)))) from area #16
 (NOT (EATS (C3 C7))) from br pt #16
 (U X ((A (X)) -> (U Y ((P (Y)) -> (EATS (X Y)))) OR
 (U Y (((A (Y)) AND (M (Y X))) AND (E Z ((EATS (Y Z)) AND (P (Z)))) -> (EATS (X Y))))) from PREMISES
 (U X ((F (X)) -> (A (X)))) from PREMISES
 #17 ((A (C3)) -> ((U Y ((P (Y)) -> (EATS (C3 Y)))) OR (U Y
 (((A (Y)) AND (M (Y C3))) AND (E Z ((EATS (Y Z)) AND (P (Z)))) -> (EATS (C3 Y))))) from area #16
 (NOT (A (C3))) from br pt #17
 #18 ((F (C3)) -> (A (C3))) from area #16
 (NOT (F (C3))) from br pt #18
 Branch Closed -> Going back to next branch

 (A (C3)) from br pt #18
 Branch Closed -> Going back to next branch

 #18 ((U Y ((P (Y)) -> (EATS (C3 Y)))) OR (U Y (((A (Y)) AND
 (M (Y C3))) AND (E Z ((EATS (Y Z)) AND (P (Z)))) -> (EATS (C3 Y))))) from br pt #17
 (U Y ((P (Y)) -> (EATS (C3 Y)))) from br pt #18
 #19 ((P (C7)) -> (EATS (C3 C7))) from area #18
 (NOT (P (C7))) from br pt #19
 Branch Closed -> Going back to next branch

 (EATS (C3 C7)) from br pt #19
 Branch Closed -> Going back to next branch

 (U Y (((A (Y)) AND (M (Y C3))) AND (E Z ((EATS (Y Z))
 AND (P (Z)))) -> (EATS (C3 Y))))) from br pt #18
 AND (P (Z)))) -> (EATS (C3 C4)) from area #18
 ((EATS (C4 Z)) AND (P (Z)))) from br pt #19
 #19 (((A (C4)) AND (M (C4 C3))) AND (E Z ((EATS (C4 Z))
 AND (P (Z)))) -> (EATS (C3 C4)) from area #18
 #20 (NOT ((A (C4)) AND (M (C4 C3))) AND (E Z
 ((EATS (C4 Z)) AND (P (Z)))) from br pt #19
 #21 (NOT ((A (C4)) AND (M (C4 C3)))) from br pt #20
 (NOT (A (C4))) from br pt #21
 (U X ((B (X)) -> (A (X)))) from PREMISES
 #22 ((B (C4)) -> (A (C4))) from area #21
 (NOT (B (C4))) from br pt #22
 Branch Closed -> Going back to next branch

 (A (C4)) from br pt #22
 Branch Closed -> Going back to next branch

 (NOT (M (C4 C3))) from br pt #21
 (U Y (((B (C4)) AND (F (Y))) -> (M (C4 Y)))) from
 #22 (((B (C4)) AND (F (C3))) -> (M (C4 C3))) from
 area #21
 #23 (NOT ((B (C4)) AND (F (C3)))) from br pt #22
 (NOT (B (C4))) from br pt #23
 Branch Closed -> Going back to next branch

 (NOT (F (C3))) from br pt #23
 Branch Closed -> Going back to next branch

 (M (C4 C3)) from br pt #22
 Branch Closed -> Going back to next branch

 (NOT (E Z ((EATS (C4 Z)) AND (P (Z)))) from br
 pt #20
 (U Z (NOT ((EATS (C4 Z)) AND (P (Z)))) from
 area #20
 #21 (NOT ((EATS (C4 C8)) AND (P (C8)))) from area #20
 (NOT (EATS (C4 C8))) from br pt #21
 Branch Closed -> Going back to next branch

 (NOT (P (C8))) from br pt #21

Branch Closed -> Going back to next branch

(EATS (C3 C4)) from br pt #19

(U X ((B (X)) -> (A (X)))) from PREMISES

(U X (NOT (E Y (((A (X)) AND (A (Y))) AND

(EATS (X Y))) AND (U Z ((G (Z)) -> (EATS (Y Z)))))) from area #0

(NOT (E Y (((A (C3)) AND (A (Y))) AND (EATS (C3 Y)))

AND (U Z ((G (Z)) -> (EATS (Y Z)))))) from area #19

(U Y (NOT (((A (C3)) AND (A (Y))) AND (EATS (C3 Y)))

AND (U Z ((G (Z)) -> (EATS (Y Z)))))) from area #19

#20 (NOT (((A (C3)) AND (A (C4))) AND (EATS (C3 C4)))

AND (U Z ((G (Z)) -> (EATS (C4 Z)))) from area #19

#21 (NOT (((A (C3)) AND (A (C4))) AND (EATS (C3 C4)))

from br pt #20

(U X ((F (X)) -> (A (X)))) from PREMISES

#22 (NOT ((A (C3)) AND (A (C4)))) from br pt #21

(NOT (A (C3))) from br pt #22

#23 ((F (C3)) -> (A (C3))) from area #21

(NOT (F (C3))) from br pt #23

Branch Closed -> Going back to next branch

(A (C3)) from br pt #23

Branch Closed -> Going back to next branch

(NOT (A (C4))) from br pt #22

#23 ((B (C4)) -> (A (C4))) from area #19

(NOT (B (C4))) from br pt #23

Branch Closed -> Going back to next branch

(A (C4)) from br pt #23

Branch Closed -> Going back to next branch

(NOT (EATS (C3 C4))) from br pt #21

Branch Closed -> Going back to next branch

(NOT (U Z ((G (Z)) -> (EATS (C4 Z)))) from br

(E Z (NOT ((G (Z)) -> (EATS (C4 Z)))) from

pt #20

area #20

(NOT ((G (C10)) -> (EATS (C4 C10)))) from area #20

(G (C10)) from area #20

(NOT (EATS (C4 C10))) from area #20

#21 ((P (C10)) -> (EATS (C4 C10))) from area #7

(NOT (P (C10))) from br pt #21

(U X ((G (X)) -> (P (X)))) from PREMISES

#22 ((G (C10)) -> (P (C10))) from area #21

(NOT (G (C10))) from br pt #22

Branch Closed -> Going back to next branch

(P (C10)) from br pt #22

Branch Closed -> Going back to next branch

(EATS (C4 C10)) from br pt #21

Branch Closed -> Going back to next branch

(NOT (P (C7))) from br pt #16

Branch Closed -> Going back to next branch

(EATS (C2 C3)) from br pt #14

Branch Closed -> Going back to next branch

(U Y (((A (Y)) AND (M (Y C4))) AND (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (C4 Y)))

from br pt #7

#8 (((A (C6)) AND (M (C6 C4))) AND (E Z ((EATS (C6 Z)) AND (P (Z)))) -> (EATS (C4 C6)))

from area #7

#9 (NOT (((A (C6)) AND (M (C6 C4))) AND (E Z ((EATS (C6 Z)) AND (P (Z)))))) from br pt #6

#10 (NOT ((A (C6)) AND (M (C6 C4)))) from br pt #9
 (NOT (A (C6))) from br pt #10.
 (U X ((S (X)) -> (A (X)))) from PREMISES

#11 ((S (C6)) -> (A (C6))) from area #10
 (NOT (S (C6))) from br pt #11
 Branch Closed -> Going back to next branch

(A (C6)) from br pt #11
 Branch Closed -> Going back to next branch

(NOT (M (C6 C4))) from br pt #10
 (U X (U Y (((S (X)) AND (B (Y))) -> (M (X Y))))) from PREMISES
 (U Y (((S (C6)) AND (B (Y))) -> (M (C6 Y)))) from area #10

#11 (((S (C6)) AND (B (C4))) -> (M (C6 C4))) from area #10

#12 (NOT ((S (C6)) AND (B (C4)))) from br pt #11
 (NOT (S (C6))) from br pt #12
 Branch Closed -> Going back to next branch

(NOT (B (C4))) from br pt #12
 Branch Closed -> Going back to next branch

(M (C6 C4)) from br pt #11.
 Branch Closed -> Going back to next branch

(NOT (E Z ((EATS (C6 Z)) AND (P (Z)))))) from br pt #9
 (U Z (NOT ((EATS (C6 Z)) AND (P (Z)))))) from area #9

#10 (NOT ((EATS (C6 C9)) AND (P (C9)))) from area #9
 (NOT (EATS (C6 C9))) from br pt #10
 Branch Closed -> Going back to next branch

(NOT (P (C9))) from br pt #10.
 Branch Closed -> Going back to next branch

(EATS (C4 C6)) from br pt #8.
 Branch Closed -> Going back to next branch

-----End of Tree-----

Appendix A-4

```
(prove '(
  (U x ((W(x)) -> (A(x))))
  (U x ((F(x)) -> (A(x))))
  (U x ((B(x)) -> (A(x))))
  (U x ((C(x)) -> (A(x))))
  (U x ((S(x)) -> (A(x))))
  (E x (W(x)))
  (E x (F(x)))
  (E x (B(x)))
  (E x (C(x)))
  (E x (S(x)))
  (E x (G(x)))
  (U x ((G(x)) -> (P(x))))
  (U x ((A(x)) -> ((U y ((P(y)) -> (Eats(x y))))
    or
    (U y (((A(y))
      and
      (M(y x))
      and
      (E z ((Eats(y z)) and (P(z))))
      -> (Eats(x y)))))))
  (U x (U y (((C(x)) and (B(y))
    -> (M(x y))))))
  (U x (U y (((S(x)) and (B(y))
    -> (M(x y))))))
  (U x (U y (((B(x)) and (F(y))
    -> (M(x y))))))
  (U x (U y (((F(x)) and (W(y))
    -> (M(x y))))))
  (U x ((W(x)) -> ((U y ((F(y)) -> (not (Eats(x y))))
    and
    (U y ((G(y)) -> (not (Eats(x y)))))))
  (U x ((B(x)) -> ((U y ((S(y)) -> (Eats(x y)))
    and
    (U y ((S(y)) -> (not (Eats(x y)))))))
  (U x ((C(x)) -> (E y ((P(y)) and (Eats(x y))))))
  (U x ((S(x)) -> (E y ((P(y)) and (Eats(x y))))))
  (E x (E y ((A(x))
    and
    (A(y))
    and
    (U z ((G(z)) -> ((Eats(x y)) and (Eats(y z))))))))
```

PROVED

(pp-tree2)

Dump of proof --> Number of closing-nodes = 48.

(NOT (E X (E Y (((A (X)) AND (A (Y))) AND (U Z ((G (Z)) -> ((EATS (X Y)) AND (EATS (Y Z)))))))) from CONCLUSION

(E X (W (X))) from PREMISES

(W (C2)) from area #0.

(U X ((W (X)) -> ((U Y ((F (Y)) -> (NOT (EATS (X Y)))) AND (U Y ((G (Y)) -> (NOT (EATS (X Y))))))) from PREMISES

(E X (B (X))) from PREMISES

(B (C4)) from area #0.

(E X (C (X))) from PREMISES

(C (C5)) from area #0.

(E X (S (X))) from PREMISES

(S (C6)) from area #0.

(E X (G (X))) from PREMISES

(G (C7)) from area #0.

(E X (F (X))) from PREMISES

(F (C3)) from area #0.

#1 ((W (C2)) -> ((U Y ((F (Y)) -> (NOT (EATS (C2 Y)))) AND (U Y ((G (Y)) -> (NOT (EATS (C2 Y))))))) from area #0
 (NOT (W (C2))) from br pt #1
 Branch Closed -> Going back to next branch

((U X ((B (X)) -> ((U Y ((C (Y)) -> (EATS (X Y)))) AND (U Y ((S (Y)) -> (NOT (EATS (X Y))))))) from PREMISE
 ((U Y ((F (Y)) -> (NOT (EATS (C2 Y)))) AND (U Y ((G (Y)) -> (NOT (EATS (C2 Y)))))) from br pt #1
 (U Y ((G (Y)) -> (NOT (EATS (C2 Y)))) from area #1

#2 ((B (C4)) -> ((U Y ((C (Y)) -> (EATS (C4 Y)))) AND (U Y ((S (Y)) -> (NOT (EATS (C4 Y)))))) from area #1
 (NOT (B (C4))) from br pt #2
 Branch Closed -> Going back to next branch

((U Y ((C (Y)) -> (EATS (C4 Y)))) AND (U Y ((S (Y)) -> (NOT (EATS (C4 Y)))))) from br pt #2
 (U Y ((F (Y)) -> (NOT (EATS (C2 Y)))) from area #1
 (U Y ((B (Y)) -> (NOT (EATS (C4 Y)))) from area #2
 (U X ((G (X)) -> (P (X)))) from PREMISES
 (U X ((C (X)) -> (E Y ((P (Y)) AND (EATS (X Y)))))) from PREMISES

#3 ((C (C6)) -> (E Y ((P (Y)) AND (EATS (C6 Y)))) from area #2
 (NOT (C (C6))) from br pt #3
 Branch Closed -> Going back to next branch

(E Y ((P (Y)) AND (EATS (C6 Y)))) from br pt #3
 ((P (C8)) AND (EATS (C6 C8))) from area #3
 (P (C8)) from area #3

#4 ((S (C6)) -> (NOT (EATS (C4 C6)))) from area #2
 (NOT (S (C6))) from br pt #4
 Branch Closed -> Going back to next branch

(U X ((S (X)) -> (E Y ((P (Y)) AND (EATS (X Y)))))) from PREMISES
 (NOT (EATS (C4 C6))) from br pt #4

#5 ((S (C6)) -> (E Y ((P (Y)) AND (EATS (C6 Y)))) from area #4
 (NOT (S (C6))) from br pt #5
 Branch Closed -> Going back to next branch

(E Y ((P (Y)) AND (EATS (C6 Y)))) from br pt #5
 ((P (C9)) AND (EATS (C6 C9))) from area #5
 (EATS (C6 C9)) from area #5
 (P (C9)) from area #5
 (U X ((A (X)) -> ((U Y ((P (Y)) -> (EATS (X Y)))) OR (U Y (((A (Y)) AND (M (Y X))) AND
 (E Z ((EATS (Y Z)) AND (P (Z)))) -> (EATS (X Y)))))) from PREMISES
 (U X ((B (X)) -> (A (X)))) from PREMISES

#6 ((A (C4)) -> ((U Y ((P (Y)) -> (EATS (C4 Y)))) OR (U Y (((A (Y)) AND (M (Y C4))) AND (E Z
 ((EATS (Y Z)) AND (P (Z)))) -> (EATS (C4 Y)))))) from area #5
 (NOT (A (C4))) from br pt #6

#7 ((B (C4)) -> (A (C4))) from area #5
 (NOT (B (C4))) from br pt #7
 Branch Closed -> Going back to next branch

(A (C4)) from br pt #7
 Branch Closed -> Going back to next branch

#7 ((U Y ((P (Y)) -> (EATS (C4 Y)))) OR (U Y (((A (Y)) AND (M (Y C4))) AND (E Z ((EATS (Y Z))
 AND (P (Z)))) -> (EATS (C4 Y)))) from br pt #6
 (U Y ((P (Y)) -> (EATS (C4 Y)))) from br pt #7

#8 ((P (C8)) -> (EATS (C4 C8))) from area #7
 (NOT (P (C8))) from br pt #8
 Branch Closed -> Going back to next branch

(EATS (C4 C8)) from br pt #8

#9 ((G (C7)) -> (P (C7))) from area #2
 (NOT (G (C7))) from br pt #9
 Branch Closed -> Going back to next branch

(P (C7)) from br pt #9
 (U X ((B (X)) -> (A (X)))) from PREMISES

#10 ((B (C4)) -> (A (C4))) from area #9

(NOT (B (C4))) from br pt #10
Branch Closed -> Going back to next branch

(A (C4)) from br pt #10.
(U X ((A (X)) -> ((U Y ((P (Y)) -> (EATS (X Y)))) OR (U Y (((A (Y)) AND
(M (Y X))) AND (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (X Y)))) from PREMISES
(U X ((W (X)) -> (A (X)))) from PREMISES

#11 ((A (C2)) -> ((U Y ((P (Y)) -> (EATS (C2 Y)))) OR (U Y (((A (Y)) AND
(M (Y C2))) AND (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (C2 Y)))) from area #10
(NOT (A (C2))) from br pt #11.

#12 ((W (C2)) -> (A (C2))) from area #10
(NOT (W (C2))) from br pt #12
Branch Closed -> Going back to next branch

(A (C2)) from br pt #12
Branch Closed -> Going back to next branch

#12 ((U Y ((P (Y)) -> (EATS (C2 Y)))) OR (U Y (((A (Y)) AND (M (Y C2))) AND
(E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (C2 Y)))) from br pt #11

(U Y ((P (Y)) -> (EATS (C2 Y)))) from br pt #12
#13 ((G (C7)) -> (NOT (EATS (C2 C7)))) from area #1
(NOT (G (C7))) from br pt #13.
Branch Closed -> Going back to next branch

(NOT (EATS (C2 C7))) from br pt #13.
#14 ((P (C7)) -> (EATS (C2 C7))) from area #12
(NOT (P (C7))) from br pt #14
Branch Closed -> Going back to next branch

(EATS (C2 C7)) from br pt #14.
Branch Closed -> Going back to next branch.

(U X (U Y (((B (X)) AND (F (Y)) -> (M (X Y)))))) from PREMISES
#13 ((F (C3)) -> (NOT (EATS (C2 C3)))) from area #2.
(NOT (F (C3))) from br pt #13.
Branch Closed -> Going back to next branch

(U Y (((A (Y)) AND (M (Y C2))) AND (E Z ((EATS (Y Z)) AND (P (Z))))))
-> (EATS (C2 Y)))) from br pt #12.
(NOT (EATS (C2 C3))) from br pt #13.
#14 (((A (C3)) AND (M (C3 C2))) AND (E Z ((EATS (C3 Z)) AND (P (Z)))))) ->
(EATS (C2 C3))) from area #13.
#15 (NOT (((A (C3)) AND (M (C3 C2))) AND (E Z ((EATS (C3 Z)) AND
(P (Z)))))) from br pt #14.

#16 (NOT ((A (C3)) AND (M (C3 C2)))) from br pt #15
(NOT (A (C3))) from br pt #16.
(U X ((F (X)) -> (A (X)))) from PREMISES
#17 ((F (C3)) -> (A (C3))) from area #16.
(NOT (F (C3))) from br pt #17
Branch Closed -> Going back to next branch

(A (C3)) from br pt #17
Branch Closed -> Going back to next branch

(NOT (M (C3 C2))) from br pt #16
(U X (U Y (((F (X)) AND (W (Y)) -> (M (X Y)))))) from PREMISES
(U Y (((F (C3)) AND (W (Y)) -> (M (C3 Y)))) from area #16
#17 ((F (C3)) AND (W (C2))) -> (M (C3 C2))) from area #16
#18 (NOT ((F (C3)) AND (W (C2)))) from br pt #17
(NOT (F (C3))) from br pt #18
Branch Closed -> Going back to next branch

(NOT (W (C2))) from br pt #18.
Branch Closed -> Going back to next branch

(M (C3 C2)) from br pt #17.
Branch Closed -> Going back to next branch

(NOT (E Z ((EATS (C3 Z)) AND (P (Z))))) from br pt #16
(U Z (NOT ((EATS (C3 Z)) AND (P (Z))))) from area #16
(U X ((A (X)) -> ((U Y ((P (Y)) -> (EATS (X Y))))) OR
(U Y (((A (Y)) AND (M (Y X))) AND (E Z ((EATS (Y Z)) AND (P (Z))))) -> (EATS (X Y))))) from PREMISES
#16 (NOT ((EATS (C3 C8)) AND (P (C8)))) from area #16
(NOT (EATS (C3 C8))) from br pt #16.
(U X ((F (X)) -> (A (X)))) from PREMISES
#17 ((A (C3)) -> ((U Y ((P (Y)) -> (EATS (C3 Y))))) OR (U Y
(((A (Y)) AND (M (Y C3))) AND (E Z ((EATS (Y Z)) AND (P (Z))))) -> (EATS (C3 Y))))) from area #16
(NOT (A (C3))) from br pt #17
#18 ((F (C3)) -> (A (C3))) from area #16
(NOT (F (C3))) from br pt #18
Branch Closed -> Going back to next branch

(A (C3)) from br pt #18.
Branch Closed -> Going back to next branch

#18 ((U Y ((P (Y)) -> (EATS (C3 Y))))) OR (U Y (((A (Y)) AND
(M (Y C3))) AND (E Z ((EATS (Y Z)) AND (P (Z))))) -> (EATS (C3 Y))))) from br pt #17
(U Y ((P (Y)) -> (EATS (C3 Y))))) from br pt #18
#19 ((P (C8)) -> (EATS (C3 C8))) from area #18
(NOT (P (C8))) from br pt #19
Branch Closed -> Going back to next branch

(EATS (C3 C8)) from br pt #19.
Branch Closed -> Going back to next branch

(U Y (((A (Y)) AND (M (Y C3))) AND (E Z ((EATS (Y Z))
AND (P (Z))))) -> (EATS (C3 Y))))) from br pt #18
(P (Z))))) -> (EATS (C3 C4)) from area #18.
Z)) AND (P (Z))))) from br pt #19

#19 (((A (C4)) AND (M (C4 C3))) AND (E Z ((EATS (C4 Z)) AND
#20 (NOT (((A (C4)) AND (M (C4 C3))) AND (E Z ((EATS (C4

#21 (NOT ((A (C4)) AND (M (C4 C3)))) from br pt #20
(NOT (A (C4))) from br pt #21
Branch Closed -> Going back to next branch

(NOT (M (C4 C3))) from br pt #21
(U Y ((B (C4)) AND (F (Y)) -> (M (C4 Y)))))

from area #12.

area #21

#22 (((B (C4)) AND (F (C3))) -> (M (C4 C3))) from

#23 (NOT ((B (C4)) AND (F (C3)))) from br pt #22
(NOT (B (C4))) from br pt #23
Branch Closed -> Going back to next branch

(NOT (F (C3))) from br pt #23.
Branch Closed -> Going back to next branch

(M (C4 C3)) from br pt #22.
Branch Closed -> Going back to next branch

(NOT (E Z ((EATS (C4 Z)) AND (P (Z))))) from br

(U Z (NOT ((EATS (C4 Z)) AND (P (Z))))) from

#21 (NOT ((EATS (C4 C8)) AND (P (C8)))) from area #20.
(NOT (EATS (C4 C8))) from br pt #21.
Branch Closed -> Going back to next branch

(NOT (P (C8))) from br pt #21.
Branch Closed -> Going back to next branch

pt #20

area #20

((G (Z)) -> ((EATS (X Y)) AND (EATS (Y Z)))))) from area #0.
 -> ((EATS (C3 Y)) AND (EATS (Y Z)))))) from area #19
 -> ((EATS (C3 Y)) AND (EATS (Y Z)))))) from area #19.
 ((EATS (C3 C4)) AND (EATS (C4 Z)))))) from area #19

(EATS (C3 C4)) from br pt #19
 (U X (NOT (E Y (((A (X)) AND (A (Y))) AND (U Z
 (NOT (E Y (((A (C3)) AND (A (Y))) AND (U Z ((G (Z))
 (U Y (NOT (((A (C3)) AND (A (Y))) AND (U Z ((G (Z))
 (NOT (((A (C3)) AND (A (C4))) AND (U Z ((G (Z)) ->
 #20 (NOT ((A (C3)) AND (A (C4)))) from br pt #20
 (NOT (A (C3))) from br pt #21
 (U X ((F (X)) -> (A (X)))) from PREMISES
 #22 ((F (C3)) -> (A (C3))) from area #21
 (NOT (F (C3))) from br pt #22
 Branch Closed -> Going back to next branch
 (A (C3)) from br pt #22
 Branch Closed -> Going back to next branch
 (NOT (A (C4))) from br pt #21
 Branch Closed -> Going back to next branch
 (NOT (U Z ((G (Z)) -> ((EATS (C3 C4)) AND
 (E Z (NOT ((G (Z)) -> ((EATS (C3 C4)) AND
 (NOT ((G (C12)) -> ((EATS (C3 C4)) AND
 (G (C12)) from area #20.
 #21 (NOT ((EATS (C3 C4)) AND (EATS (C4 C12)))) from
 (NOT (EATS (C3 C4))) from br pt #21
 Branch Closed -> Going back to next branch
 (NOT (EATS (C4 C12))) from br pt #21.
 (U X ((G (X)) -> (P (X)))) from PREMISES
 #22 ((P (C12)) -> (EATS (C4 C12))) from area #7
 (NOT (P (C12))) from br pt #22.
 #23 ((G (C12)) -> (P (C12))) from area #21
 (NOT (G (C12))) from br pt #23.
 Branch Closed -> Going back to next branch
 (P (C12)) from br pt #23
 Branch Closed -> Going back to next branch
 (EATS (C4 C12)) from br pt #22
 Branch Closed -> Going back to next branch
 (NOT (P (C8))) from br pt #16
 Branch Closed -> Going back to next branch
 (EATS (C2 C3)) from br pt #14.
 Branch Closed -> Going back to next branch
 (U Y (((A (Y)) AND (M (Y C4))) AND (E Z ((EATS (Y Z)) AND (P (Z)))))) -> (EATS (C4 Y)))
 from br pt #7.
 #8 (((A (C6)) AND (M (C6 C4))) AND (E Z ((EATS (C6 Z)) AND (P (Z)))))) -> (EATS (C4 C6))
 from area #7.
 #9 (NOT (((A (C6)) AND (M (C6 C4))) AND (E Z ((EATS (C6 Z)) AND (P (Z)))))) from br pt #8
 #10 (NOT ((A (C6)) AND (M (C6 C4)))) from br pt #9.
 (NOT (A (C6))) from br pt #10.
 (U X ((S (X)) -> (A (X)))) from PREMISES
 #11 ((S (C6)) -> (A (C6))) from area #10.
 (NOT (S (C6))) from br pt #11.

Branch Closed -> Going back to next branch

(A (C6)) from br pt #11

Branch Closed -> Going back to next branch

(NOT (M (C6 C4))) from br pt #10

(U X (U Y (((S (X)) AND (B (Y))) -> (M (X Y))))) from PREMISES

(U Y (((S (C6)) AND (B (Y))) -> (M (C6 Y)))) from area #10.

#11 ((S (C6)) AND (B (C4))) -> (M (C6 C4)) from area #10

#12 (NOT ((S (C6)) AND (B (C4)))) from br pt #11.

(NOT (S (C6))) from br pt #12.

Branch Closed -> Going back to next branch

(NOT (B (C4))) from br pt #12.

Branch Closed -> Going back to next branch

(M (C6 C4)) from br pt #11

Branch Closed -> Going back to next branch

(NOT (E Z ((EATS (C6 Z)) AND (P (Z))))) from br pt #9.

(U Z (NOT ((EATS (C6 Z)) AND (P (Z))))) from area #9

#10 (NOT ((EATS (C6 C9)) AND (P (C9)))) from area #9

(NOT (EATS (C6 C9))) from br pt #10.

Branch Closed -> Going back to next branch

(NOT (P (C9))) from br pt #10.

Branch Closed -> Going back to next branch

(EATS (C4 C6)) from br pt #8.

Branch Closed -> Going back to next branch

-----End of Tree-----

Appendix A-5

... Schubert's Steamroller with missing premises
 ... demonstration of nontheorem detection

```
(prove '(
  (U x ((W(x)) -> (A(x))))
  (U x ((F(x)) -> (A(x))))
  (U x ((B(x)) -> (A(x))))
  (U x ((C(x)) -> (A(x))))
  (U x ((S(x)) -> (A(x))))
  (E x (W(x)))
  (E x (F(x)))
  (E x (B(x)))
  (E x (C(x)))
  (E x (S(x)))
  (E x (G(x)))
  (U x ((G(x)) -> (P(x))))
  (U x ((A(x)) -> ((U y ((P(y)) -> (Eats(x y))))
    or
    (U y (((A(y))
      and
      (M(y x))
      and
      (E z ((Eats(y z)) and (P(z))))
      -> (Eats(x y)))))))
  (U x (U y (((C(x)) and (B(y)))
    -> (M(x y)))))
  (U x (U y (((S(x)) and (B(y)))
    -> (M(x y)))))
  (U x (U y (((B(x)) and (F(y)))
    -> (M(x y)))))
  (U x ((B(x)) -> ((U y ((C(y)) -> (Eats(x y)))
    and
    (U y ((S(y)) -> (not (Eats(x y))))))))
  (U x ((C(x)) -> (E y ((P(y)) and (Eats(x y))))))
  (U x ((S(x)) -> (E y ((P(y)) and (Eats(x y))))))
  '(E x (E y (((A(x))
    and
    (A(y))
    and
    (Eats(x y))
    and
    (U z ((G(z)) -> (Eats(y z))))))))
  Theorem is not true --> No more formulae to decompose
```

END

|22·06·88

FIN