Analysis of the Use of Semantic Trees in Automated Theorem Proving

by

Mohammed A. Almulla

School of Computer Science
McGill University
February 10, 1995

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in Computer Science

Abstract

Semantic trees have served as a theoretical tool for confirming the unsatisfiability of clauses in first-order predicate logic, but it has seemed impractical to use them in practice. In this thesis we experimentally investigated the practicality of generating semantic trees for proofs of unsatisfiability. We considered two ways of generating semantic trees. First, we looked at semantic trees generated using the canonical enumeration of atoms from the Herbrand base of the given clauses. Then, we considered semantic trees generated by selectively choosing the atoms from the Herbrand base using an improved semantic tree generator, AISTG. A comparison was made between the two approaches using the theorems from the "Stickel Test Set". In underlying the practicality of using semantic tree generators as mechanical theorem provers, another comparison was made between the AISTG and a resolution-refutation theorem prover "The Great Theorem Prover".

Résumé

Les arbres sémantiques ont servi d'outils théoriques pour confirmer l'insatisfiabilité des propositions logiques à prédicat du premier ordre. Cependant, il est connue que leur utilisation n'est pas très pratique. Dans cette thèse, nous étudions expérimentallement la question de la possibilité d'utiliser la génération d'arbres sémantiques afin de prouver l'insatisfiabilité des propositions logiques.

Nous avons ainsi considéré deux facons de générer des arbres sémantiques. Dans un premier temps, nous nous sommes concentrés sur les arbres semantiques qui ont été généres en utilisant l'énumeration canonique des atomes à partir de la base de Herbrand de la proposition donnée.

Puis, dans un deuxième temps, nous avons considéré les arbres sémantiques qui sont génerés en choisissant de manière sélective les atomes de la base Herbrand. Une comparaison à été faite entre les deux approches en utilisant les théorèmes de " l'ensemble de tests Stickel".

Enfin, pour souligner l'importance de la praticabilité de l'utilisation des génerateurs d'arbres sémantiques en tant que théorèmes prouveurs, nous avons comparé GAAS[1], avec la procédure de résolution-réfutation du "Grand Théorème Prouveur".

[1]Un Générateur Amelioré d'Arbres Sémantiques

Statement of Originality

Although all work herein that is not otherwise cited represents an original and distinct contribution to the study of semantic trees and their role in automated theorem proving, Professor Monroe M. Newborn nonetheless deserve special recognition. It is he who opened my eyes to the power of semantic trees when he asked me the following question: *"Can semantic trees efficiently prove unsatisfiability?"*

Another motive behind this recognition is due to the AISTG[2] program, which we implemented for the development of this research. Parts of this program are based on The Great Theorem Prover of Newborn [Newborn1]. Other parts of the program including the control strategies, the construction of the semantic trees, and the extraction of resolution-refutation proofs from the closed semantic trees are the sole responsibility of the author.

[2]An Improved Semantic Tree Generator

Acknowledgements

The pursuit of research requires the continued support of many people and establishments over a number of years. I gratefully acknowledge this everlasting support and extend my deep appreciation to:

- My supervisor, Professor Monroe M. Newborn, to whom I owe a sincere thanks for continuous guidance and concern. His efforts certainly have gone far beyond the normal duties of a research supervisor.
- The government of Kuwait represented by his highness the Emir of Kuwait, his Crown Prince, and Kuwait University, which granted me the financial means to undertake graduate studies.
- My advisor at the Embassy of the State of Kuwait, Cultural Division, Kuwait University office, Dr. Abdel-Rahim S. Abdussalam, for all the help and support he embraced me with in times of need.
- Azzedine Boukerche and John Kozlowski for their assistance with this dissertation. Their technical expertise helped me to refine many of my intuitions into concrete results.

Finally, any success that I enjoy has always been a testament to my wonderful family, in particular my parents. It is to my wife Fatima H. Bu-Shahri and my two children, Jassem and Farah Almulla, that I dedicate this work with all my love.

Thesis Outline

This dissertation attempts a comprehensive treatment of semantic trees in automated theorem proving. Of course, semantic trees can be investigated from a number of distinct perspectives, such as artificial intelligence, mathematics, linguistics and cognitive science. This thesis adopts the first perspective, focusing on the role of semantic trees in the automation of theorem proving. Since it is infeasible to incorporate everything, the thesis tries to provide a careful balance between the depth and breadth of the presented material. An important technical goal of our study is to provide sufficient information so that the reader can comprehend and possibly implement most (if not all) of the included algorithms. References are provided at the end of the dissertation in case of a need for further exploration of relevant matters. The structure of this thesis is divided into five chapters as described below.

Chapter 1: devoted to theory, methods, and applications of semantic trees in automated theorem proving. The chapter begins by specifying the objectives of this research. Next, it presents a survey of previous attempts at using semantic trees to confirm the unsatisfiability of sets of clauses. Then, it explores some linear Herbrand's proof procedures. This is followed by a formal definition of the terminology necessary for the reader to be familiar with the subject on hand. Lastly, the chapter describes the set of theorems used for measuring the performance of the semantic tree generators under investigation in this study. Our goal in Chapter 1 is to provide both the background and the motivation for our research.

Chapter 2: presents semantic trees generated using the canonical enumeration of atoms from the Herbrand base. We will call such trees *canonical semantic trees*. The chapter begins with an extended inspection of the Herbrand universe and of the Herbrand base of a set of clauses, accompanied by examples. Next, the chapter introduces a system of using semantic trees to prove the unsatisfiability of sets of clauses. The system includes building closed semantic trees from given resolution-refutation proofs, building canonical semantic trees using Herbrand's procedure, and extracting resolution-refutation proofs from closed semantic trees. Then, the chapter compares the performance of a canonical semantic tree generator and The Great Theorem Prover on the Stickel Test Set. Our goal in Chapter 2 is to investigate the practicality of generating semantic trees for proving the unsatisfiability of sets of clauses.

Chapter 3: suggests methods for improving the performance of semantic tree generators as mechanical theorem provers. The influence of these methods on a semantic tree generator is examined using the Stickel Test Set, and the semantic tree generator is, thus, described as *improved*. The tables and graphs appearing in this chapter illustrate this influence. Our goal in Chapter 3 is to improve the practicality of generating semantic trees for proofs of unsatisfiability.

Chapter 4: discusses the development and implementation of *AISTG: An Improved Semantic Tree Generator*, developed particularly for the purpose of this study. The chapter is concerned with programming aspects of the AISTG. It describes the structure, flow of control, modules and layouts of the program; it also specifies the capabilities and limitations of the program. In addition, a guided tour by the title *"Using the AISTG Program"* is included in Chapter 4. The chapter ends with a comparison made between the AISTG and a resolution-refutation theorem prover. Our goal in Chapter 4 is to appreciate the complexity of the AISTG as a pragmatic semantic tree generator which is capable of proving reasonably hard theorems.

Chapter 5: discusses semantic tree generation as an alternative method for proving the unsatisfiability of sets of clauses as opposed to resolution-refutation. The chapter presents theorems for which more efficient proofs were obtained by using semantic tree generators; this is in contrast to those proofs obtained by resolution-refutation theorem provers. Conversely, the chapter demonstrates examples of theorems for which resolution-refutation proofs are much more desirable. Our goal in Chapter 5 is to facilitate and encourage both the use of our AISTG program and the reliance on semantic tree generation to assist in automated theorem proving.

Chapter 6: devoted to the conclusion of this investigation. First, this chapter summarizes the findings of the previous chapters. Second, it makes links between our research and (i) other related research areas, and (ii) open research problems in automated theorem proving related to our work. These links both occur in the form of offered suggestions for further improvements.

Contents

List of Figures

List of Tables

1 Introduction

The fundamental theorem of Herbrand has great significance in symbolic logic. It is a base for most modern proof procedures [Nossum1, Loveland5] including the resolution principle of Robinson, the connection graph procedure of Kowalski, the matrix reduction method of Prawitz, the model elimination procedure of Loveland, and the compactness procedure of Nossum. It has also qualified semantic tree generators as mechanical theorem provers. According to this theorem, in order to test whether a set S of clauses is unsatisfiable, one must consider all interpretations over all possible domains. However, for most theorems of any difficulty, it is hard to evaluate all interpretations over all possible domains in order to establish unsatisfiability. It would be simpler to fix on a special domain, such that the set S is unsatisfiable if and only if S is false under all interpretations over this domain. Fortunately, there does exist such a domain, and it is called the *Herbrand universe* of S. Nonetheless, as the Herbrand universe can possibly be infinite, interpretations over this domain should be organized in some systematic way. This can easily be achieved by using *semantic trees* [Manna1].

Aside from being a base for other proof procedures, Herbrand's theorem

suggested a pragmatic refutation procedure best known as *Herbrand's procedure*. That is, for a given unsatisfiable set S of clauses, if there is a mechanical procedure that can successively generate sets S'_1, S'_2, ... of ground instances of clauses in S and can successively test S'_1, S'_2, ... for unsatisfiability, then (as guaranteed by Herbrand's theorem) this procedure can eventually reach an N such that S'_N is unsatisfiable [Chang1].

In spite of its simplicity, Herbrand's procedure has one major drawback; it requires the generation of sets S'_1, S'_2, ... of ground instances of clauses. For most cases, this sequence grows exponentially [Chang1]. Consequently, many researchers have agreed that semantic trees can be used to confirm the unsatisfiability of sets of clauses, but they have felt that they are impractical for actually determining the unsatisfiability of those sets [Chang1, Loveland2, Manna1, Nilsson3].

In this thesis, we investigate semantic tree generators as mechanical theorem provers in both theory and practice. Not only do we identify the problems causing their poor performance, but we also propose solutions to those problems. In addition, we develop the general theory of semantic trees, concentrating on those cases of semantic tree construction in which we have found improvements in performance of the semantic tree generators. In addition, we present An Improved Semantic Tree Generator *(AISTG)* as a practical theorem prover for first-order predicate theorems.

In the remainder of this chapter, we bring to bear the potential role of semantic trees in automated theorem proving. First, we define the problem of generating semantic trees to prove the unsatisfiability of sets of clauses, and state the objectives of this research. Next, we shed light on the basic knowledge underlying this problem. Then, we identify and formally define the terminology necessary. Finally, we close the chapter by a description of the set of theorems used for testing the semantic tree generators under investigation in this study.

1.1 Problem Definition

Traditionally, semantic trees have been used as a theoretical tool for confirming the unsatisfiability of sets of clauses in propositional and first-order predicate logic [Chang2, Hsiang1, Kowalski4, Slagle3] although it has seemed impractical to use them for detecting the unsatisfiability of those sets [Chang1, Loveland2, Manna1, Nilsson3]. Our objectives in this thesis are not only to measure but to improve their practicality. To do this, we implemented a semantic tree generator and ran some tests with it. The first test generated semantic trees using the canonical enumeration of atoms from the Herbrand base. Other tests are due to methods proposed for improving the performance of the semantic tree generator. The methods are:

1. Filtering the Herbrand base,

2. Selectively choosing atoms from the Herbrand base,

3. Incorporating advice-taking into generating semantic trees.

It will be seen in the sequel that the first method occasionally speeds up the construction of the semantic tree. The second method customizes the semantic tree, by controlling the manner by which the Herbrand base atoms participate in building the semantic tree. The last method is best justified by the following remark [LiMin1]:

> The immediate benefit of machine learning would be to enable AI programs to improve their performance automatically over time. For example, a chess program can improve its game plan against its opponent through playing. A robot can recognize a particular kind of object more accurately through repeated presentation of the object image. At a more fundamental level, a machine with a clearly demonstrated ability to learn would answer the question whether machines can exhibit true intelligence. Without this capability, a computer system can not reason beyond the limit of its programmed intelligence. In fact, an entity can hardly be called intelligent unless it can learn.
>
> Fu LiMin

In principle, these three methods accommodate desirable properties which, in turn, improved the performance of the semantic tree generator. That is, the semantic tree generator proved larger and more difficult theorems in contrast to the canonical semantic tree generator. Additionally, a comparison was made between the performance of the improved semantic tree generator *(AISTG)* and the performance of a resolution-refutation theorem prover, *The Great Theorem Prover* [Newborn1]. It is by improving the practicality of generating semantic trees for proving theorems that we wish to introduce semantic tree generators as practical theorem provers.

1.2 History and Background

In June 11th, 1930, J. Herbrand was granted his doctorate in mathematics with highest honors at the *École Normale Supérieure* in Paris. In his thesis, Herbrand presented a famous theorem later known as *Herbrand's fundamental theorem* [Herbrand1]. This theorem had grown to become one of the bases in symbolic logic nowadays. The use of Herbrand's fundamental theorem allowed a reduction of first-order predicate logic tasks to truth table checkable propositional logic tasks (in a perhaps infinite-ary logic). This theorem was originally concerned with valid rather than unsatisfiable theorems, and was argued purely syntactically. Later on a more semantic (i.e. model-theoretic) approach was employed, which greatly eased the complexity of argument [Manna1].

Based on Herbrand's fundamental theorem an apparently natural process for checking tautologyhood by the name of *Herbrand's procedure* had appeared. Gilmore was one of the first researchers to implement Herbrand's procedure on a computer in 1960 [Gilmore1]. Gilmore's program managed to prove a few simple theorems, but it encountered difficulties with most other first-order logic theorems. Careful studies of his program revealed that Gilmore's method of testing for the unsatisfiability was inefficient [Chang1]. To overcome this inefficiency, Davis and Putnam [Putnam1] introduced a more efficient method for testing the unsatisfiability of a set of ground clauses. Their method consisted of

four rules, the principal one of which was to break a difficult and long theorem into small and simple cases and then to prove the theorem by considering each case separately. However, their effect was still not enough. Many theorems in first-order predicate logic still could not be proved by computers in reasonable amounts of time. Other recognized attempts to use Herbrand's procedure in automated theorem proving are due to [Kowalski4, Chang1, Hsiang1, Slagle3]. The difference in objectives or the incomplete success of these attempts is the motivation for our research.

A major breakthrough was made by Robinson in 1965 when he introduced the *resolution principle* [Robinson5]. Proof procedures based on the resolution principle are much more efficient than are any of the earlier Herbrand proof procedures. Since the introduction of the resolution principle, several refinements have been suggested in attempts to further increase efficiency. Some of these refinments are *Semantic Resolution* [Slagle3, Robinson6, Kowalski4], *Lock Resolution* [Boyer1], *Linear Resolution* [Loveland3, Loveland4, Loveland5, Luckham1, Anderson1, Reiter1, Kowalski3], *Unit Resolution* [Wos5, Chang3], and the *Set-of-Support Strategy* [Wos4].

From the discovery of Robinson's resolution principle up until now, a wide gap has existed between the performance of Herbrand-dependent proof procedures and resolution-dependent proof procedures. In this work, we bring these two far ends to a closer range. Our aim behind this is to enlarge the underestimated role of semantic trees in automated theorem proving.

1.3 Terminology

A clear exposition of the necessary preliminaries can be found in [Chang1, Fermuller1, Robinson3]. Nonetheless, in this section we provide formal definitions for the basic notions of predicate logic, fundamental concepts in theorem proving (such as substitution, unification, and resolution), Herbrand semantics, and semantic trees. Additional terminology is introduced in later chapters whenever this aids in the understanding of our definitions and proofs.

1.3.1 Terms, Literals, and Clauses

Concerning the language of predicate calculus [Fermuller1], we assume that there is an infinite supply of variable symbols, constant symbols, function symbols and predicate symbols. Moreover, we assume that each function and predicate symbol is associated with some fixed arity. We call a function or a predicate symbol *unary* if it is of arity 1, *binary* if it is of arity 2, and in general n-ary if it is of arity n. A constant symbol is a function symbol of arity 0.

In first-order predicate calculus, a statement is called a well-formed formula (wff). A wff is interpreted as making a statement about some domain of discourse [Newborn1]. The syntax of wffs usually requires:

Definition 1.3.1 *Logical operators* are: & [conjunction], | [disjunction], $\sim$ [negation], $\Rightarrow$ [implication], and $\Longleftrightarrow$ [if and only if].

Definition 1.3.2 *Quantifiers* are: $\forall$ [universal quantifier] and $\exists$ [existential quantifier].

Definition 1.3.3: A *term* is defined recursively as follows:

1. Each variable and each constant is a term.
2. If $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is also a term.

If a term t is of the form $f(t_1, \ldots, t_n)$ we call it functional; the set of arguments of t, args(t), is the set $\{t_1, \ldots, t_n\}$; f is called the *leading function symbol* of t.

Definition 1.3.4: If $t_1, \ldots, t_n$ are terms and P denotes an n-ary predicate symbol, then $A = P(t_1, \ldots, t_n)$ is an *atom*; the set of arguments of A, args(A), is the set $\{t_1, \ldots, t_n\}$; P is called the *leading predicate symbol* of t.

Definition 1.3.5: A *literal* is either an atom or an atom preceded by the negation sign "$\sim$".

Definition 1.3.6: A *expression* is either a term or a literal.

Definition 1.3.7 A *well-formed formula* (wff) is defined recursively as follows:

1. A literal is a wff.

2. If w is a wff, then so is the negation of w, $\sim w$.

3. If w and v are wffs, then so are $w|v$, $w \ \& \ v$, $w \Rightarrow v$, and $w \Longleftrightarrow v$.

4. If w is a wff, then, for any variable x, so are $\forall x{:}w$ and $\exists x{:}w$.

A statement in predicate calculus is a *wff*. A *theorem* is a set of wffs some of which are axioms and the rest are the conclusion. Often, deciding what axioms to use and deciding the exact wording of the axioms and the conclusion is the most difficult part of the theorem-proving procedure [Newborn1]. Therefore, some theorems provers (including The Great Theorem Prover) do not attempt to find a proof of a theorem expressed as a set of wffs. Instead, they first compile the wffs to a simpler form called clauses. Then, using these clauses as input, they attempt to find a proof. An algorithm for converting wffs to clause form is given in [Newborn1, Nilsson2]. In our work, we are going to consider only formulas given in clause form. For those formulas which are given as wffs, we use the **COMPILE** procedure of The Great Theorem Prover for their conversion.

Definition 1.3.8: A *clause* is a finite disjunction of zero or more literals.

Definition 1.3.9: An expression or a clause is called *ground* if no variables occur in it. We call it *constant free* if no constants occur in it, and *function free* if it does not contain function symbols.

The null clause, denoted by "[ ]", is a clause of zero literals. Throughout this work when we speak of sets of clauses, we always mean finite sets of clauses.

Definition 1.3.10: The *term depth* of a term t denoted as term_depth(t) is defined by :

(a) If t is a variable or a constant, then term_depth(t) = 0.

(b) If t = f(t_1, ..., t_n), where f is an n-ary function symbol, then term_depth(t) = 1 + max { term_depth(t_i) | $1 \leq i \leq n$ }.

(c) The term depth of a literal L(t_1, t_2, ..., t_n), where L is an n-ary predicate symbol, is defined as: term_depth(L) = max { term_depth(t_i) | $1 \leq i \leq n$ }.

(d) The term depth of a clause C = L_1 | L_2 | ...| L_n is defined as: term_depth(C) = max { term_depth(L_i) | $1 \leq i \leq n$ }.

(e) The term depth of a set S = { C_1, C_2, ..., C_n } is defined as: term_depth(S) = max { term_depth(C_i) | $1 \leq i \leq n$ }.

Example 1: If L_1 = P(x,f(f(y))), L_2 = Q(f(x)) and C = L_1 | ~L_2, then term_depth(L_1) = 2, term_depth(L_2) = 1, term_depth(C) = max { 1, 2 } = 2.

1.3.2 Substitution, Unification, and Resolution

A short-term goal of this research is to prove theorems in propositional and first-order predicate calculus by using semantic tree generators. A basic notion in theorem proving which is required for achieving this goal is the concept of substitution, for which we use the definitions given in [Chang1].

Definition 1.3.11: A *substitution* is a finite set of the form $\{t_1/v_1, t_2/v_2, \ldots, t_n/v_n\}$, where every v_i is a variable, every t_i is a term different from v_i, and no two elements in the same set have the same variable after the stroke symbol.

Definition 1.3.12: When $t_1, t_2, \ldots, t_n$ are ground terms, the substitution is called a *ground substitution.* The substitution that consists of no elements is called the *empty substitution.*

Definition 1.3.13: Let $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ be a substitution and E be an expression. Then Eθ is an expression obtained from E by replacing simultaneously each occurrence of the variable v_i, for $i = 1, \ldots, n$, in E by the term t_i. Eθ is called an instance of E.

1t E be an expression and σ a substitution. The *application* of σ to E is defined as follows:

(a) If E is a variable, then Eσ is σ(E).

(b) If E is a constant, then Eσ = E.

(c) Otherwise, E is of the form X(t_1, ..., t_n), where X is an n-ary function symbol or predicate symbol. In either case, Eσ = X($t_i\sigma$, ..., $t_n\sigma$).

(d) If L is a literal, then Lσ is defined to be the application of σ to the atom of L.

(e) If C is a clause, then Cσ = {Eσ |$\forall$ E $\in$ C }.

Definition 1.3.14: Let E_1 and E_2 be expressions, then $E_1 \leq_s E_2$ - read: E_1 is *more general* than E_2 - if and only if there exists a substitution σ such that $E_1\sigma = E_2$. Similarly, if C and D are clauses, $C \leq_s D$ if and only if there exists a substitution σ such that C$\sigma \subseteq$ D. In this case we may say, in accordance with the usual resolution terminology, that C *subsumes* D.

Definition 1.3.15: A set of expressions M is *unifiable* by a substitution σ if and only if $E_i\sigma = E_j\sigma$ for all E_i and $E_j \in$ M. σ is called the *most general unifier (mgu)* of M if and only if for every other unifier ϑ of M: $\sigma \leq_s \vartheta$.

1.3.3 Resolvents

For resolvents, we retain the original definition in [Robinson5], which combines *factorization* and *binary resolution.*

Definition 1.3.16: A *factor* of a clause C is a clause Cσ, where σ is a mgu of some C' $\subseteq$ C. In case the number of literals in Cσ is less than the number of literals in C, we call the factor *non-trivial.*

Definition 1.3.17: If C and D are two clauses and M and N are literals of C and D respectively, such that M $\cup$ $\sim$N is unifiable by the mgu σ, then clause

E = (C - M)σ $\cup$ (D - N)σ is a *binary resolvent* of C and D. The atom A of (M $\cup$ $\sim$N)σ is called the *resolved atom.*

1.3.4 Herbrand Semantics

For Herbrand semantics, we refer to the original terminology in [Herbrand2].

Definition 1.3.18: Let HU_0 be the set of all constants appearing in S. If no constant appears in S, then $HU_0 = \{ a \}$, where a is an arbitrary constant. For i = 1, 2, ..., let HU_i be the union of HU_{i-1} and the set of all terms of the form $f(t_1, \ldots, t_n)$ for all functions occurring in S, where each t_j, for $j = 1, 2, \ldots, n$, is a member of the set HU_{i-1}. Each HU_i is called the *i-level constant set* of S, and HU_∞ is called the *Herbrand universe* of S [Chang1].

Definition 1.3.19: Let HB_0 be the set of all ground literals of the form $P(t_1, t_2, \ldots, t_n)$ for all predicates in S, where each t_j, for $j = 1, 2, \ldots, n$, is a member of HU_0. For i = 1, 2, ..., define HB_i to be the union of HB_{i-1} and the set of all ground literals of the form $P(t_1, t_2, \ldots, t_n)$ for all predicates occurring in S, where each t_j, for $j = 1, 2, \ldots, n$, is a member of the set HU_i (the *i*-level constant set). Each HB_i is called the *i-level predicate set*, and HB_∞ is called the *Herbrand base* of S. Elements of the Herbrand base are called *atoms.*

Both the Herbrand universe and the Herbrand base of S are either finite or countably infinite sets, and thus we can refer to the *i-th* element in either set by an enumeration algorithm.

Definition 1.3.20: Arbitrarily, we order all functions appearing in the given set of clauses with respect to their arity. The *canonical enumeration of elements from the Herbrand universe*, denoted by HU, is, then, a recursive enumeration [Stoll1] of the terms from the Herbrand universe.

Definition 1.3.21: The *canonical enumeration of atoms from the Herbrand base*, denoted by HB, is a recursive enumeration of all ground literals for all predicates in S obtained by using the canonical enumeration of elements from

the Herbrand universe.

Example 2: Consider the theorem **Wos12** (from the Stickel Test Set [Stickel2]). The symbols e and a are constants.

Axioms:

01. p(e,x,x)
02. p(g(x),x,e)
03. ~p(x,y,z) | ~p(y,u,v) | ~p(z,u,w) | p(x,v,w)
04. p(x,y,f(x,y))
05. ~p(x,y,z) | ~p(y,u,v) | ~p(x,v,w) | p(z,u,w)
06. r(x,x)
07. ~r(x,y) | r(y,x)
08. ~r(x,y) | ~r(y,z) | r(x,z)
09. ~p(x,y,z) | ~p(x,y,u) | r(z,u)
10. ~p(z,u,x) | p(z,u,y) | ~r(x,y)
11. ~p(z,x,u) | p(z,y,u) | ~r(x,y)
12. ~p(x,z,u) | p(y,z,u) | ~r(x,y)
13. ~r(x,y) | r(f(z,x),f(z,y))
14. ~r(x,y) | r(f(x,z),f(y,z))
15. ~r(x,y) | r(g(x),g(y))
16. p(x,e,x)
17. ~p(x,g(y),z) | ~o(x) | ~ o(y) | o(z)
18. p(x,g(x),e)
19. ~r(x,y) | ~o(x) | o(y)
20. o(a)

Negated Theorem:

21. ~o(e)

HU(Wos12) = { a, e, g(a), g(e), f(a,a,), f(a,e), f(e,a), f(e,e), g(g(a)), g(g(e)), g(f(a,a)), g(f(a,e)), g(f(e,a)), g(f(e,e)), f(a,g(a)), f(a,g(e)), f(a,f(a,a)), f(a,f(a,e)), f(a,f(e,a)), f(a,f(e,e)), f(e,g(a)), f(e,g(e)), f(e,f(a,a)), f(e,f(a,e)), f(e,f(e,a)), f(e,f(e,e)), f(g(a),a), ... }.

HB(Wos12) = { o(a), r(a,a), p(a,a,a), o(e), r(a,e), r(e,a), r(e,e), p(a,a,e), p(a,e,a), p(a,e,e), p(e,a,a), p(e,a,e), p(e,e,a), p(e,e,e), o(g(a)), r(a,g(a)), r(e,g(a)), r(g(a),a), r(g(a),e), r(g(a),g(a)), p(a,a,g(a)), p(a,e,g(a)), p(a,g(a),a), p(a,g(a),e), p(a,g(a),g(a)), p(e,a,g(a)), p(e,e,g(a)), p(e,g(a),a), p(e,g(a),e), p(e,g(a),g(a)), p(g(a),a,a), ... }.

Example 3: As a second example, consider **Starkey103** (also from the Stickel Test Set [Stickel2]). The symbols A and B are constants.

Axioms:

01. ~S(x,y) | ~M(z,x) | M(z,y)
02. S(x,y) | M(F(x,y),x)
03. S(x,y) | ~M(F(x,y),y)
04. S(x,y) | ~E(x,y)
05. S(y,x) | ~E(x,y)
06. ~S(x,y) | ~S(y,x) | E(x,y)
07. ~M(u,z) | M(u,x) | M(u,y) | ~UN(x,y,z)
08. ~M(u,x) | M(u,z) | ~UN(x,y,z)
09. ~M(u,y) | M(u,z) | ~UN(x,y,z)
10. M(G(x,y,z),z) | M(G(x,y,z),x) | M(G(x,y,z),y) | UN(x,y,z)

11. ~M(G(x,y,z),x) | ~M(G(x,y,z),z) | UN(x,y,z)
12. ~M(G(x,y,z),y) | ~M(G(x,y,z),z) | UN(x,y,z)
13. UN(A,A,B)

Negated Theorem:

14. ~E(B,A)

HU(Starkey103) = { A, B, F(A,A), F(A,B), F(B,A), F(B,B), G(A,A,A), G(A,A,B), G(A,B,A), G(A,B,B), G(B,A,A), G(B,A,B), G(B,B,A), G(B,B,B), F(A,F(A,A)), F(A,F(A,B)), ... }.

HB(Starkey103) = { E(A,A), M(A,A), S(A,A), UN(A,A,A), E(A,B), E(B,A), E(B,B), M(A,B), M(B,A), M(B,B), S(A,B), S(B,A), S(B,B), UN(A,A,B), UN(A,B,A), UN(A,B,B), UN(B,A,A), UN(B,A,B), UN(B,B,A), UN(B,B,B), E(A,F(A,A)), E(B,F(A,A)), E(F(A,A),A), E(F(A,A),B), E(F(A,A),F(A,A)), ... }.

From the construction of a Herbrand universe (and thus a Herbrand base) of a set of clauses it can be seen that *infiniteness* of the universe (and thus the base) is inevitable whenever a function symbol is introduced in one of the clauses [Wang1].

Definition 1.3.22: An *interpretation* of S consists of a nonempty domain D and of an assignment of "values" to each constant symbol, function symbol, and predicate symbol occurring in S as follows:

1. To each constant, we assign an element in D.
2. To each n-ary function symbol, we assign a mapping from D^n to D.
3. To each n-ary predicate symbol, we assign a mapping from D^n to { TRUE, FALSE }.

Based on an interpretation for S, a value of TRUE or FALSE can be assigned to each atom of S, and in turn to each clause of S. A value of TRUE is assigned to clause C if the disjunction of values of the atoms of C is TRUE. Otherwise, a value of FALSE is assigned. If C is assigned a value of TRUE, we say the interpretation *satisfies* C.

The set S is said to be *satisfiable* if and only if there exists an interpretation over the Herbrand universe for which all clauses of S are assigned the value TRUE. Such an interpretation is called a *model* for S.

Definition 1.3.23: The set S is said to be *unsatisfiable* if and only if for every possible interpretation over the Herbrand universe there is at least one clause that has the value FALSE.

To prove a theorem using the technique of proof by contradiction, it is sufficient to show that S, the set consisting of the set of axioms and the negated conclusion, is unsatisfiable.

Definition 1.3.24: A *Herbrand interpretation* for S, denoted by HI(S), is a subset of the Herbrand base HB(S) for which the truth value TRUE is assigned to all atoms of HI(S) and the truth value FALSE is assigned to all atoms not in HI(S).

Herbrand's Theorem: *A set* S *of clauses is unsatisfiable if and only if there is a finite unsatisfiable set* S′ *of ground instances of clauses of* S.

Proof: Suppose there is a finite unsatisfiable set S′ of ground instances of clauses in S; suppose further that I′ is an interpretation of S′. The lifting lemmas [Newborn1] justifies extending the transformation of the set S′ to the more general set S. Thus, it is safe to assume that every interpretation I of S contains an interpretation I′ of S′. Since every interpretation I of S contains an interpretation I′ of S′, if I′ falsifies S′, then I must also falsify S′. However, S′ is falsified by every interpretation I′. Consequently, S′ is falsified by every interpretation I of S. Therefore, S is falsified by every interpretation of S. Hence, S is unsatisfiable. To show the "only if" statement, we establish the equivalent contrapositive statement: if every finite subset of S′ is satisfiable, then S is satisfiable. If every finite subset of S′ is satisfiable, then S′ itself is satisfiable. Therefore S′ has a ground model found by taking the TRUE literals of any model of S′. A direct translation of this ground model to a

model for S shows that S is also satisfiable [Almulla1]. Q.E.D.

Definition 1.3.25: Herbrand's theorem suggested a procedure, known as *Herbrand's procedure*, for proving the unsatisfiability of sets of clauses. For a given set S of clauses, we can generate successively ground instances of the clauses of S and test successively whether their conjunction is unsatisfiable. By Herbrand's theorem, if S is unsatisfiable, the procedure will detect it after a finite number of steps. Otherwise, the procedure might never terminate.

Definition 1.3.26: Atoms in the Herbrand base that neither they nor their complements resolve with any clause in S are called *useless atoms*. Whereas, atoms in the Herbrand base that either they or their complements, but not both, resolve with some clauses in S are called *unnecessary atoms*.

Both useless and unnecessary atoms are of no help in detecting the unsatisfiability of S, and are unnecessary to use when growing semantic trees. A Herbrand base of the set S with all useless and unnecessary atoms eliminated is called a *filtered Herbrand base*. Moreover, a semantic tree generated from a filtered Herbrand base is called a *filtered semantic tree*.

Example 4: Consider the following theorem. If S = {P(x), ~P(a) | Q(f(a)), ~Q(f(x))}, then HU(S) = {a, f(a), f(f(a)), ...}, and HB(S) = {P(a), Q(a), P(f(a)), Q(f(a)), P(f(f(a))), Q(f(f(a))), ...}. The atoms P(a) and ~P(a) can be seen to resolve with the clauses of S. Therefore, P(a) is neither useless nor unnecessary. The atom Q(a) does not resolve with any clause in S nor does its complement. Therefore, Q(a) is useless. Except for Q(f(a)), the remaining atoms in the Herbrand base are all unnecessary. Thus, the *Filtered Herbrand Base:* FHB(S) = {P(a), Q(f(a))}.

1.3.5 Semantic Trees

The definition of a semantic tree for clauses in first-order predicate logic can be found in [Robinson1, Kowalski3, Hayes1]. In our presentation, we assign

clauses to the non-terminal nodes of the semantic tree (in addition to th eterminal node as others have done) in order to assist in obtaining a semantic tree proof.

Definition 1.3.27: A *semantic tree* of a set of clauses is a downward growing binary tree. The branches of the tree are labelled with atoms from the Herbrand base and their negation. Let the atoms from the Herbrand base be ordered as $hb_1, hb_2, \ldots, hb_j, \ldots$. A node N in a semantic tree is said to be at depth j if and only if it is j nodes away from the root of the tree along some path. Left branches leading to nodes at depth j are labelled with hb_j; right branches are labelled with $\sim hb_j$. Each node N is assigned a set of clauses as follows:

1. If N is the root of the tree, assign all base clauses to it.

2. If N is not the root of the tree, then it has some parent M. The clauses assigned to N depend both on the set A of clauses assigned to the nodes on the path from the root node to M and on the Herbrand atom or its negation – in either case denoted by literal L – labelling the branch leading from M to N and is determined as follows. For each clause C in A, place in node N all resolvents of C and L and all resolvents of the resolvents with L until no more resolvents are generated.

The mechanism of semantic trees permits insight into the process of establishing completeness for the first-order predicate proof procedures. It also provides a direct link with the notion of resolution itself. Whether the Herbrand base of an unsatisfiable set S of clauses is finite or countably infinite, only a finite subset of it is necessary for constructing a closed semantic tree (see Definition 1.3.30) for that set. Moreover, the order in which the atoms appear in the enumeration of this subset dictates the size and shape of the closed semantic trees.

Definition 1.3.28: A *canonical semantic tree* is a semantic tree in which

the branches at level i are labelled with the i-th atom from the canonical enumeration of atoms from the Herbrand base or its negation.

Different enumerations of atoms from the Herbrand base yield different semantic trees. One of these enumerations corresponds to the canonical semantic tree.

Definition 1.3.29: A node N in a semantic tree is called a *failure node* if the null clause is assigned to it.

A failure node in a semantic tree is indicated by a solid node (•) when it appears in the tree and is labelled with the number of the base clause C and the indices of the literal La or literals La, Lb, ... of C that were resolved away to yield a failure. Other nodes on the path to N record where other literals of C, if there were other literals, were resolved away.

Definition 1.3.30: If every path in a semantic tree beginning at the root terminates at a failure node, the tree is called a *closed semantic tree* and contains a finite number of nodes above the failure nodes.

We now have a way of confirming the unsatisfiability of a set S of clauses; the confirmation involves building a closed semantic tree for S. If S is unsatisfiable, then, by Herbrand's theorem, there is a finite subset K of the Herbrand base such that every semantic tree T for K is closed for S. However, determining the unsatisfiability of S by constantly generating semantic trees and efficiently testing them for closure has been considered awkward and impractical [Chang1, Loveland1, Manna1, Nilsson2].

Definition 1.3.31: A *vine* is a finite binary tree in which each node is either a leaf or is immediately above some leaf. A node N of a vine is a *bottom-leaf* if N is below every node of the vine which is not a leaf.

It should be noted that a vine which has more than one node has exactly two bottom-leaves.

1.4 The Stickel Test Set

It is never an easy task to find a large number of theorems with suitable accessibility, variety, and difficulty. For the testing of semantic tree generators investigated in this thesis we followed [Stickel2] in using the set of theorems appeared in the Wilson and Minker study [Wilson1]. This set was later known as the *Stickel Test Set*. Two other sets of theorems satisfying the above conditions are the seventy-five problems for testing automatic theorem provers [Pelletier1] and the theorems which appeared in the automated development of Tarski's geometry [Quaife1], although for our purposes, the Stickel Test Set seemed more suitable for its flexibility, domain variety, and proof availability.

The original source of the Stickel Test Set is the Wilson and Minker study in 1976 [Wilson1]. They took theorems 1-9 from Reboh et al. [Reboh1], theorems 10-19 from Michie et al. [Michie1], theorems 20-24 from Fleisig et al. [Fleisig1], theorems 25-57 from Wos [Wos3], and theorems 58-84 from Starkey and Lawrence [Starkey1]. This last set of theorems has been used to test the Markgraf Karl Refutation Procedure connection-graph resolution theorem-proving program [Karl1]. In 1988, Stickel enlarged this set of theorems by adding 9 theorems to it from [Chang1], and used the enlarged set to test his Prolog Theorem Prover [Stickel2]. Letz et al. have also used the Stickel Test Set to test their SETHEO theorem-proving program [Letz1].

In our study of semantic trees, we used the first 84 theorems of the Stickel Test Set because they cover a wide range of theorems of varying difficulty. More importantly, they illustrate the need to dramatically prune the search space (that is, the set of all possible ways of applying resolution to the base clauses and all resolvents deduced), which make them sufficient for testing the semantic tree generators.

2

Canonical Semantic Trees in Automated Theorem Proving

Herbrand's fundamental theorem has many profound contributions in symbolic logic. It was shown in Chapter 1 how Herbrand's theorem implied a refutation procedure for proving theorems in propositional and first-order predicate logic. In this chapter, another contribution made by Herbrand's theorem is illustrated. The theorem revealed correspondence between semantic trees and resolution-refutation proof trees, and between semantic tree generators and resolution-refutation theorem provers [Wang1]. In the sequel, we demonstrate this correspondence by showing that semantic tree generators are indeed equivalent to their counterpart.

Resolution with merging is a complete deductive system for the first-order predicate calculus and is compatible with the set-of-support strategy [Andrews1]. The s-linear deductive system of Loveland (which is a restriction on resolution) is complete and, as with merging, is compatible with the set-of-support strategy [Loveland4]. The Ancestry Filtered Form (also called Linear Form) is also a complete deductive system and is compatible with the set-of-support strategy [Luckham1]. The compatibility of these deductive systems refers to the correspondence between closed semantic trees and resolution-

refutation proofs trees. The same closed semantic tree may be generated using atoms obtained from two or more resolution-refutation proofs which, in turn, are constructed by different, yet compatible, deductive systems.

Concentrating on semantic trees and their role in automated theorem proving, this chapter investigates the practicality of generating semantic trees for proofs of unsatisfiability. The chapter begins with a close look at the Herbrand universe and at the Herbrand base of a set of clauses. It underlines the reason behind avoiding the use of semantic tree generators as mechanical theorem provers. Focusing on the growth rate of the Herbrand universe and of the Herbrand base, the chapter presents mathematical formulas which reflect this extremely rapid growth. In addition, the chapter presents a system for using semantic trees in proving unsatisfiability of sets of clauses. The system includes building canonical semantic trees by Herbrand's procedure, extracting resolution-refutation proofs from closed semantic trees, and building closed semantic trees from given resolution-refutation proofs. To achieve its primary objective, the chapter ends with displaying the result of measuring and comparing the performance of a canonical semantic tree generator with The Great Theorem Prover on the Stickel Test Set.

2.1 Growth Rate Analysis of the Herbrand Universe

The major combinatorial obstacle to efficiency for Herbrand-dependent semantic tree generators is the enormous growth rates of the constant sets and of the predicate sets (see definitions 1.3.18 and 1.3.19), and hence the growth rates of the Herbrand universe and of the Herbrand base of a set of clauses [Robinson5]. They can be – and most often are – both exponential [Chang1, Hsiang1, Robinson1]. These growth rates were analyzed in some detail in [Robinson7]. Nonetheless, in estimating the efficiency of canonical semantic tree generators as mechanical theorem provers, we developed the following formulas:

Let | HU_i | denote the number of terms in HU_i. Then,

$$| HU_0 | = number\ of\ constants\ in\ the\ set\ S$$

$$| HU_1 | = | HU_0 | + \sum_{m=1}^{k} n_m * | HU_0 |^m$$

and for $i > 1$,

$$| HU_i | = | HU_{i-1} | + \sum_{m=1}^{k} n_m * (| HU_{i-1} |^m - | HU_{i-2} |^m)$$

where n_m is the number of m-ary functions, and k is the maximum number of arguments in any function.

Let us apply the above formulas to the two theorems given in Example 2 and Example 3 of Chapter 1.

Example 1: The number of constant symbols in the theorem Wos12 = 2, the number of unary function symbols in Wos12 = 1 and the number of binary function symbols in Wos12 = 1. Therefore, | HU_0 | = 2, | HU_1 | = 8, | HU_2 | = 74, | HU_3 | = 5552, ...etc.

Example 2: The number of constant symbols in Starkey103 = 2, the number of unary function symbols in Starkey103 = 0, the number of binary function symbols in Starkey103 = 1 and the number of 3-ary function symbols in Starkey103 = 1. Therefore, | HU_0 | = 2, | HU_1 | = 14, | HU_2 | = 2942, ...etc.

These values are used subsequently for estimating the growth rate of the Herbrand base of these two theorems, as it will be seen in the next section.

2.2 Growth Rate Analysis of the Herbrand Base

Let | HB_i | denote the number of atoms in HB_i. Then,

$$| HB_0 | = \sum_{m=1}^{K} N_m * | HU_0 |^m$$

$$| HB_1 | = | HB_0 | + \sum_{m=1}^{K} N_m * (| HU_1 |^m - | HU_0 |^m)$$

and for $i > 1$,

$$| HB_i | = | HB_{i-1} | + \sum_{m=1}^{K} N_m * (| HU_i |^m - | HU_{i-1} |^m)$$

where N_m is the number of m-ary predicates, and K is the maximum number of arguments in any predicate.

For completeness purposes, we apply the above formulas to Wos12 and Starkey103.

Example 1:

The number of unary predicate symbols in Wos12 = 1, the number of binary predicate symbols in Wos12 = 1 and the number of 3-ary predicate symbols in Wos12 = 1. Therefore, $| HB_0 | = 14$, $| HB_1 | = 584$, $| HB_2 | = 410774$, $| HB_3 | = 1.71 \times 10^{11}$, ... etc.

Example 2:

There are no unary predicate symbols in Starkey103. The number of binary predicate symbols in Starkey103 = 3 and the number of 3-ary predicate symbols in Starkey103 = 1. Therefore, $| HB_0 | = 20$, $| HB_1 | = 3332$, $| HB_2 | = 2.54 \times 10^{10}$, ... etc.

2.3 Building Canonical Semantic Trees

Given some axioms and a negated conclusion, a *base clause* is either a member of the axioms or a member of the clauses in the negated conclusion. Occasionally, a given set of base clauses can be simplified prior to building its canonical semantic tree. By simplifying, we mean to eliminate certain clauses from the set and or to eliminate certain literals from the clauses. These simplifications are such that the simplified set of base clauses is unsatisfiable if and only if the original set is unsatisfiable [Nilsson2]. Performing such simplifications may optimize the canonical semantic tree by trimming redundant parts of the tree. Simplifying the clauses is carried out by the following three procedures:

- **Procedure I (Uncomplemented literals removal)**: If a base clause C has a literal L that can not be resolved with any other literal in the set of base clauses, then L can be eliminated from C. The justification of this step is that L can not contribute to finding of a proof (provided that one exists).

- **Procedure II (Subsumed clauses removal)**: A clause in an unsatisfiable set of clauses which is subsumed (see Definition 1.3.14) by another clause in the set can be eliminated without affecting the unsatisfiability of the set [Newborn1]. In case of resolution-refutation, the elimination of clauses subsumed by others frequently leads to substantial reductions in the number of resolutions that need to be performed for finding a refutation [Nilsson2]. This statement can be extended to justify performing Procedure II prior to building a closed semantic tree for an unsatisfiable set of clauses.

- **Procedure III (Tautology clauses removal)**: A clause is a *tautology* precisely when it contains a pair of oppositely signed but otherwise identical literals. Such clauses can obviously be eliminated without losing refutation completeness. The justification of this step is due to the fact that any unsatisfiable set of clauses containing a tautology is still unsatisfiable after removing the tautology, and conversely. Thus, clauses such as P(f(a)) | ~P(f(a)) and P(x) | Q(y) | ~Q(y) may be eliminated.

Consider the following two examples for constructing canonical semantic trees.

Example 3: Let S_1 be the following theorem:

C1: P(x) | Q(y)

C2: ~P(a)

C3: ~Q(b)

In this case, the canonical enumeration of elements from the Herbrand universe

is finite. $HU(S_1) = \{ a, b \}$. Accordingly, the canonical enumeration of atoms from the Herbrand base is finite and is ordered as follows: $HB(S_1) = \{ P(a), Q(a), P(b), Q(b) \}$.

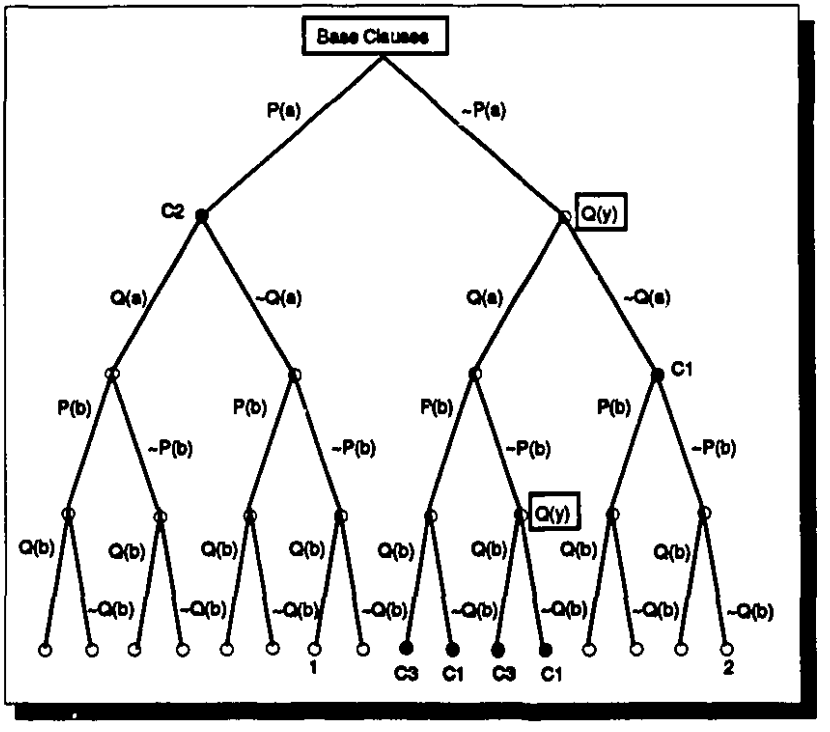

Figure 2.1: **A canonical semantic tree for S_1.**

The canonical semantic tree for the set S_1 is closed, finite and is shown in Figure 2.1. Tracing down a path from the root node to a tip node (i.e. a node at the bottom of the tree), provides one Herbrand interpretation of the set S_1 (see Definition 1.3.24). Thus, the Herbrand interpretation obtained by tracing from the root node to the tip node marked 1 in Figure 2.1 is given by the set:

$$M_1 = \{ P(a), \sim Q(a), \sim P(b), Q(b) \}$$

Such a set is a model for S_1. A model fails to satisfy a clause if there exists a ground instance of the clause (using terms from Herbrand universe) having the value FALSE, using the valuation specified by the model. Hence, the model M_1 fails to satisfy the clauses $\sim P(a)$ and $\sim Q(b)$. Similarly, the model M_2 $= \{ \sim P(a), \sim Q(a), \sim P(b), \sim Q(b) \}$ fails to satisfy the clause $P(x) \mid Q(y)$,

since the ground instance P(a) | Q(b) has the value FALSE. We can eliminate each of the 16 possible interpretations, in turn, to conclude that the set S_1 is unsatisfiable.

Example 4: Let S_2 be the following theorem:

$\sim P(x) \mid Q(x)$

$P(f(y))$

$\sim Q(f(y))$

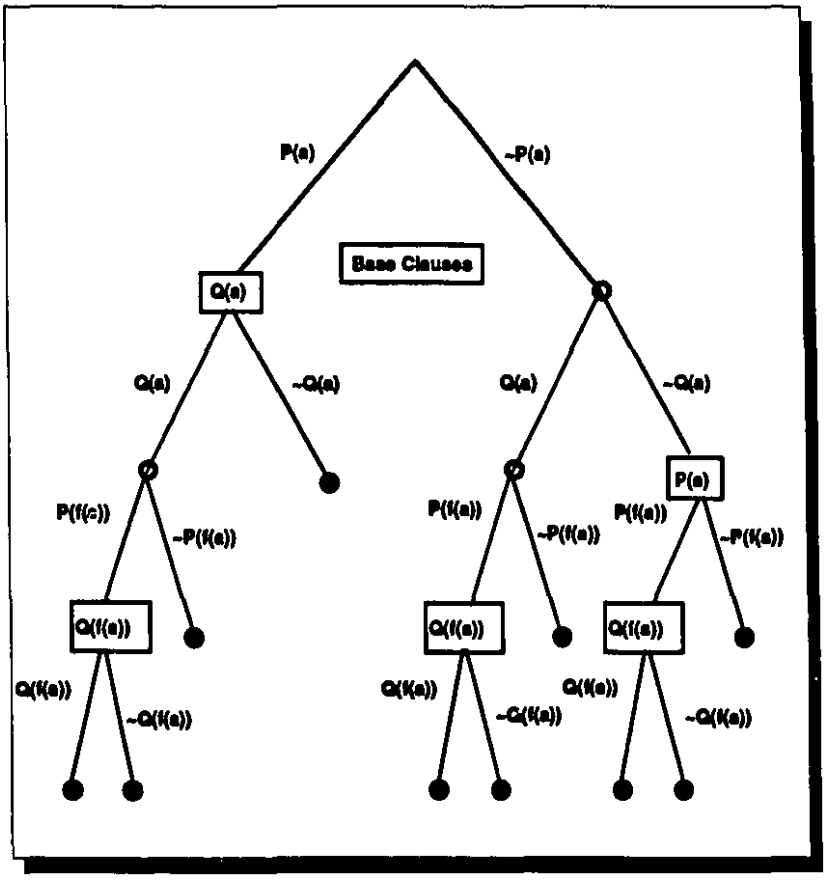

Figure 2.2: **A canonical semantic tree for S_2.**

The canonical enumeration of elements from the Herbrand universe of S_2 is: $HU(S_2) = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$. Accordingly, the canonical enumeration of atoms from the Herbrand base of S_2 is: $HB(S_2) = \{ P(a), Q(a), P(f(a), Q(f(a)), P(f(f(a))), \ldots\}$.

If the Herbrand base of a set S of clauses is countably infinite, as it for the above set S_2, then each complete Herbrand interpretation corresponds to an infinite path in the semantic tree. Nevertheless, all semantic trees of S must be closed by failure nodes, including the canonical semantic tree, if and only if the set S is unsatisfiable. The part of the canonical semantic tree above and including all failure nodes for the set S_2 is shown in Figure 2.2.

2.4 Proving Theorems Using Canonical Semantic Trees

Proof procedures can be efficient when they are used with knowledge and intelligence. However, when they are used purely mechanically (i.e. used without any programmed intelligence to reduce the search overhead), they can be and most often are inefficient. Certain proofs, however, are well-adapted to mechanical use. Resolution-refutation is the best known such proof procedure; semantic tree generation is another, though it has not had the intensive development that resolution-refutation has. Our aim here is to present the latter system in considerable detail, and to discuss its implementation. In this section, an algorithm for extracting resolution-refutation proofs from closed semantic trees is demonstrated. This algorithm will, then, be used to prove the completeness of generating semantic trees as a method for proving theorems.

A resolution-refutation proof tree is a special case of a *resolution-refutation proof graph.* In a resolution-refutation proof tree, a node in the tree serves as an input to only one other node, while in a resolution-refutation proof graph a node may serve as an input to more than one node [Newborn1]. Constructing a resolution-refutation proof graph of a theorem from the corresponding closed semantic tree is done according to the following algorithm:

1. Consider two failure nodes N1 and N2 that are siblings of node N and that fail because of clauses C1 and C2, respectively. Let L1a, L1b, ..., L1n denote the literals resolved away in C1 and L2a, L2b, ..., L2m denote

the literals resolved away in C2. If only one literal is resolved away in each clause, say L1a and L2a respectively, form the binary resolvent R = (C1L1a,C2L2a). Otherwise, if there is more than one literal resolved away in C1, say L1a, L1b, ..., L1n, first form a factor of C1, say C1', using a substitution $\theta 1$ that is a mgu of the literals L1a, L1b, ..., L1n; similarly if there is more than one literal resolved away in C2, say L2a, L2b, ..., L2m, form a factor of C2, say C2', using a substitution $\theta 2$ that is a mgu of the literals L2a, L2b, ..., L2m. Let L1a' = { L1a | L1b | ...| L1n }$\theta 1$ and L2a' = { L2a | L2b | ...| L2m }$\theta 2$. Then form the resolvent R = (C1'L1a',C2'L2a'). Each time this step is performed, one new resolvent is added to the proof and possibly one or more factors are added as well.

2. Form a new semantic tree for the enlarged set of clauses including the base clauses and all resolvents created thusfar. This new semantic tree will have at least one less node than did the previous semantic tree, failing at all the nodes that its predecessor did and failing at node N as well due to the new resolvent R added in Step (1). Often, the new resolvents will cause nodes on the path to the failure nodes of the previous semantic tree to fail in this new semantic tree. Eventually, a semantic tree will be created with only a root node and that will be a failure node. The proof will then be complete.

Example 5: Let S_3 be the following set of base clauses:

Axioms:

1. P(x)
2. ~P(a) | Q(x)

Negated Theorem:

3. ~Q(f(x))

There are two tip nodes in the closed semantic tree of S_3 appearing in Figure 2.3 (a). Resolving them together gives R4 = (3a,2b) = ~P(a). The

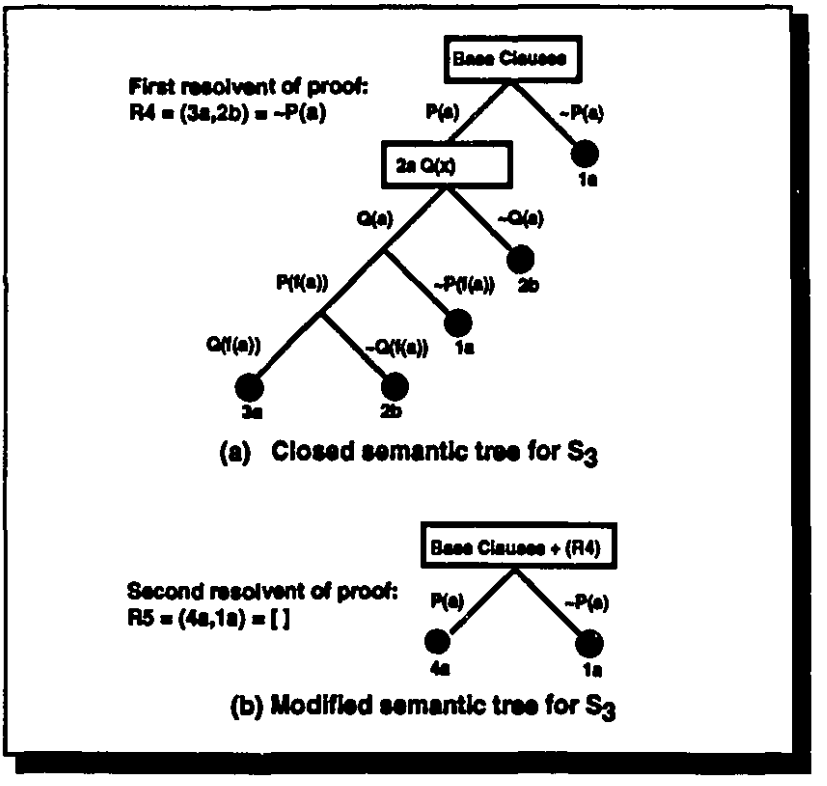

Figure 2.3: **Closed semantic trees for S_3.**

semantic tree for the enlarged set of clauses is shown in Figure 2.3 (b). The null clause falls out: R5 = (4a,1a) = [ ]. The resolution-refutation proof is presented next.

4. (3a,2b) ~P(a)
5. (4a,1a) [ ]

Example 6: Let S_4 be the following set of base clauses:

Axioms:

1. P(h(x,y),x) | Q(x,y)
2. ~P(x,y) | Q(y,x)

Negated Theorem:

3. ~Q(x,y)

There are two choices for the first step. Arbitrarily, form clause R4 = (2a,1a) = Q(x,y) | Q(x,h(x,y)) as shown in Figure 2.4 (a). Then, construct the semantic tree for the modified set of clauses containing the three base

clauses and clause R4. Again, there are two choices for the second resolvent. Arbitrarily, form clause R5 = (3a,4b) = Q(x,y) as shown in Figure 2.4 (b). Then, once again construct the semantic tree for the set of clauses containing the first three base clauses and clauses R4 and R5. Lastly, form clause R6 = (3a,5a) = [ ] as shown in Figure 2.5. The resolution-refutation proof is presented below.

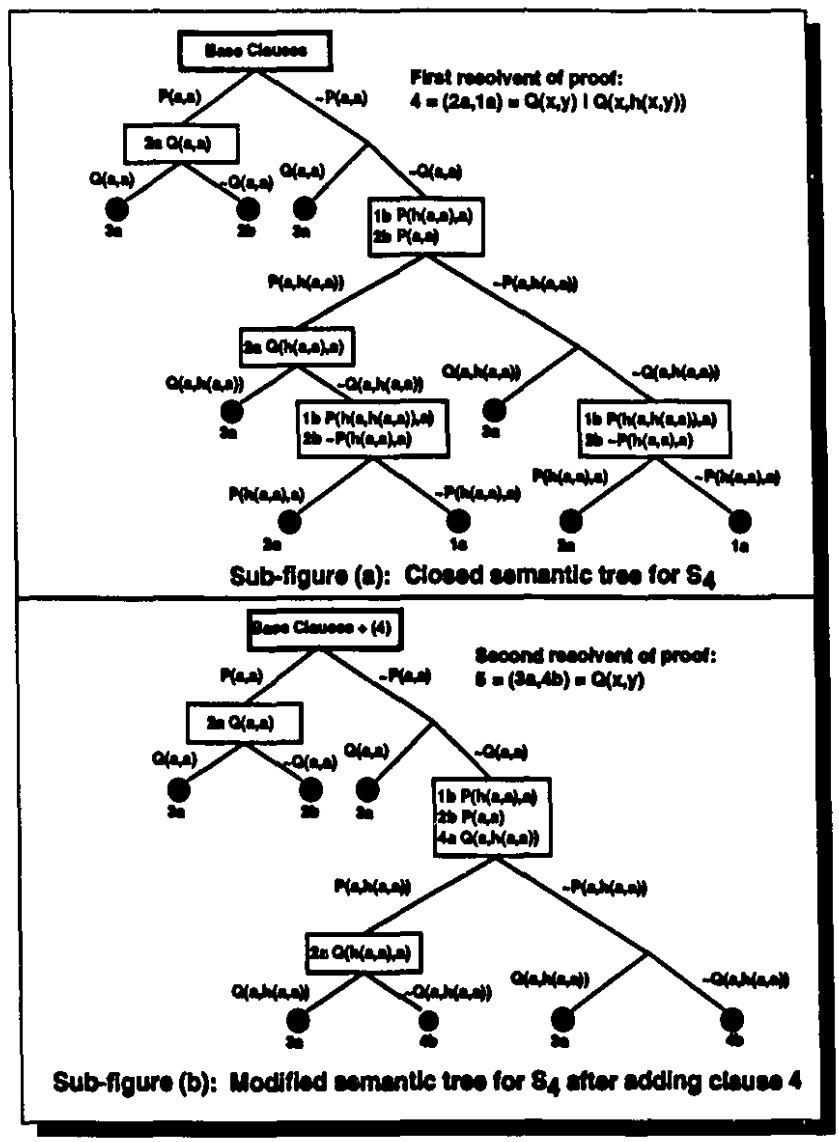

Figure 2.4: **Closed semantic trees for S_4.**

4. (2a,1a) Q(x,y) | Q(x,h(x,y))
5. (3a,4b) Q(x,y)

6. (5a,3a) [ ]

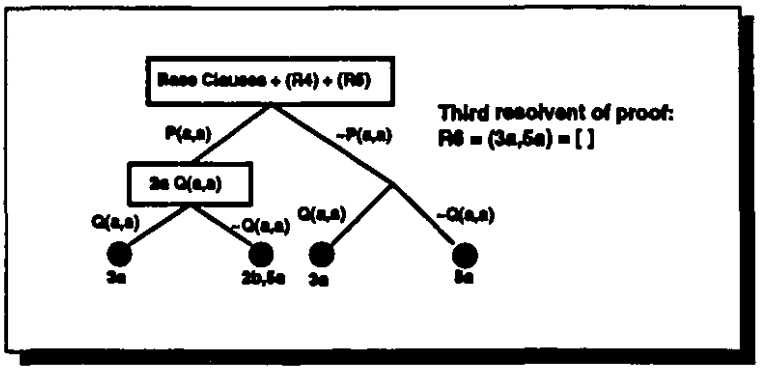

Figure 2.5: **A modified semantic tree for S_4 after adding clauses R4 and R5.**

The above algorithm can be used in proving the completeness of generating semantic trees for proving the unsatisfiability of sets of clauses.

Theorem 2.1: *A finite set* S *of clauses is unsatisfiable if and only if there is a semantic tree deduction of the null clause* [ ].

Proof: ($\Longrightarrow$) Suppose S is unsatisfiable. Then, by Herbrand's theorem there exists a finite closed semantic tree for S. If the closed semantic tree is built in such a way that (1) all useless and unnecessary atoms are filtered, and (2) every failure node in the tree is labelled with the number of the clause and index of the literal resolved away, which the Herbrand interpretation failed to satisfy, then a deduction of the null clause [ ] can be obtained from the closed semantic tree of S. Simply by repeatedly traversing the tree in a bottom-up and left-right fashion, the base clauses whose numbers label the two failure nodes sharing the same parent are resolved, thereby enlarging S by adding the resolvent to it. Eventually, a new semantic tree is built for the enlarged set S with only one node (and that is the root node) which will be a failure node. The deduction of [ ] is then obtained.

($\Longleftarrow$) Conversely, suppose there is a semantic tree deduction of [ ] from S. Let R1, R2, ..., Rk, for some positive integer k, be the resolvents obtained

by resolving the base clauses whose numbers label the failure nodes in the closed semantic tree. Assume S is satisfiable. Then, there is a model M for S. However, if a model satisfies two clauses C1 and C2, it must also satisfy any resolvent of C1 and C2. Since M satisfies S, it must also satisfy R1, R2, ..., Rk. But this a contradiction, because one of these resolvents is [ ]. Therefore, S must be unsatisfiable. Q.E.D.

2.5 The Stickel Test Set Experiment

The canonical semantic tree generator is a naive (i.e. purely mechanical) theorem prover. In this section we discuss an experiment that is meant to analyze the effectiveness of proving theorems using semantic trees. We investigate the amount of search required from the canonical semantic tree generator to prove a theorem, given an exact amount of search sufficient for the proof. The latter is obtained from a resolution-refutation proof of that theorem.

The Great Theorem Prover is a resolution-based theorem prover.It can serve as the instructional material for a course in automated theorem proving [Newborn1]. It uses two inference rules, **binary resolution** and **binary factoring**, when attempting to prove theorems. This prover was used for proving research theorems such as those in the Stickel Test Set and Tarski's geometry. In addition, it was used as a resolution-based tool for testing the abstract theories of COCOLOG [Caines1].

Before attempting to construct closed semantic trees for the theorems in the Stickel Test Set, we will examine resolution-refutation proofs of each theorem in the set as found by The Great Theorem Prover [Newborn2]. From the proof of each theorem, we extract the set A of resolved atoms A_1, A_2, ..., A_n. That set is sufficient for constructing a closed semantic tree. The term depth of A was determined from the smallest i such that all resolved atoms that are in A are in HB_i, placing an upper bound on the depth of a closed canonical semantic tree. The value of HB_{i-1} places a lower bound on this depth for the set A. It should be pointed out that the proof to most theorems is not unique

and that the set of resolved atoms, in turn, is not unique, and thus while a closed canonical semantic tree must exist with depth at most HB_i, one may in fact exist with depth less than HB_{i-1}.

Example 7: Wos12 serves as a good example to show how the set of resolved atoms are extracted from a proof, and how the minimal value of i can be determined.

The proof of Wos12 is:

22: (21a,18d) ~p(x,g(y),e) | ~o(x) | ~o(y)

23: (22a,17a) ~o(x)

24: (23a,20a) [ ]

where [ ] denotes the null clause.

When resolving Clause 23 and Clause 20 to generate Clause 24, the resolved atom is o(a). The constant a is substituted for x in Clause 23 to form 23′: ~o(a). When resolving Clause 22 and Clause 17 to give 23′: ~o(a), the resolved atom is p(a,g(a),e). The constant a is substituted for x and y in Clause 22 to form clause 22′: ~p(a,g(a),e) | ~o(a) | ~o(a). When resolving Clause 21 and Clause 18 to give Clause 22′, the resolved atom is o(e). The three resolved atoms o(a), o(e) and p(a,g(a),e) all have a term depth of 1, and thus a closed semantic tree of depth at most $| HB_1 | = 584$ can be found. Further, $| HB_0 | = 14$ tells us that a depth of at least 14 levels of the canonical semantic tree must be investigated before a closed semantic tree which corresponds to this particular proof is generated.

Table 2.1 measures the performance of a canonical semantic tree generator on the theorems in the Stickel Test Set. This table presents the name of each theorem (Column 1), the number of resolved atoms for each theorem (Column 2), the smallest value of i such that i-level predicate set contains all the resolved atoms (Column 3), the value of HB_{i-1} (Column 4), and the value of HB_i (Column 5). It can be seen that the depth of the semantic trees of

[3]The asterisk character in this column indicates that the number of generated resolvents is greater than the size of the clause database.

	Using The Great Theorem Prover				Using Canon. Sem. Tree Generator		
Theorem Name	Resolved Atoms	i	$\lvert HB_{i-1} \rvert$	$\lvert HB_i \rvert$	Proof Found	Time in seconds	Atoms [3] checked
S01burst	16	2	1.06×10^8	7.49×10^{22}	No	3900	66
S02short	8	2	8.29×10^4	6.93×10^9	No	14560	97
S03prime	8	2	78	406	Yes	19800	187
S04haspart1	4	1	810	2.06×10^{14}	No	61	275
S05haspart2	7	2	2.06×10^{14}	3.78×10^{71}	No	178	275
S06ances	6	0	0	6	Yes	1	6
S07NUM1	4	1	39	258	Yes	16810	174
S08group1	3	2	125	5.31×10^5	No	11963	63
S09group2	10	0	0	64	Yes	21631	54
S10ew1	5	0	0	5	Yes	0	5
S11ew2	3	0	0	3	Yes	0	3
S12ew3	5	0	0	5	Yes	0	5
S13rob1	8	2	8	50	Yes	56	41
S14rob2	10	0	0	64	Yes	9360	54
S15michie	3	2	64	5.06×10^4	Yes	26835	182
S16qw	3	1	1	4	Yes	0	3
S17mqw	3	1	1	4	Yes	0	3
S18DBABHP	6	3	1.20×10^9	7.28×10^{17}	No	18000	62
S19APABHP	18	4	5.50×10^{22}	3.03×10^{45}	No	290	275
S20fleisig1	11	3	50	1352	No	3405	275
S21fleisig2	11	3	50	1352	No	3501	275
S22fleisig3	13	2	738	9282	No	3720	70
S23fleisig4	8	2	1024	8.10×10^5	No	3060	42
S24fleisig5	8	2	1024	8.10×10^5	No	3420	42
S25Wos1	6	2	80	1.62×10^4	No	3948	95
S26Wos2	6	2	576	4.10×10^5	No	7200	70
S27Wos3	3	0	0	12	Yes	0	8
S28Wos4	5	2	155	7.06×10^4	Yes	1571	72
S29Wos5	7	2	576	4.10×10^5	No	4500	101
S30Wos6	8	1	80	1.44×10^4	No	7560	56
S31Wos7	6	1	36	3600	Yes	11528	74
S32Wos8	6	2	576	4.10×10^5	No	7742	135
S33Wos9	7	1	150	4.41×10^4	No	7200	130
S34Wos10	10	0	0	80	Yes	7140	70
S35Wos11	8	1	150	4.41×10^4	No	12600	105
S36Wos12	3	1	14	584	Yes	12	24
S37Wos13	5	1	14	1110	No	3600	103
S38Wos14	5	1	14	584	Yes	13	30
S39Wos15	10	2	1.44×10^4	2.20×10^5	No	3600	108
S40Wos16	6	1	14	1110	Yes	6	24
S41Wos17	6	1	39	6174	No	10265	134[4]
S42Wos18	5	1	84	1.44×10^4	No	7560	108

Table 2.1: **The Great Theorem Prover vs Canonical Semantic Tree Generator.**

	Using The Great Theorem Prover				Using Canon. Sem. Tree Generator		
Theorem Name	Resolved Atoms	i	$\lvert HB_{i-1} \rvert$	$\lvert HB_i \rvert$	Proof Found	Time in seconds	Atoms [3] checked
S43Wos19	7	1	155	2.19×10^5	No	48408	168
S44Wos20	16	1	155	2.19×10^5	No	3060	100
S45Wos21	9	2	3600	5.51×10^7	No	3060	102
S46Wos22	14	3	7.66×10^{11}	2.35×10^{24}	No	2500	47*
S47Wos23	5	1	275	4.35×10^5	No	5659	71*
S48Wos24	6	1	275	4.35×10^5	No	5663	71*
S49Wos25	6	1	1088	5.99×10^6	No	5659	71*
S50Wos26	24	2	1.72×10^5	2.22×10^{11}	No	18260	128
S51Wos27	6	1	63	7.29×10^4	No	3780	48
S52Wos28	9	1	275	3.35×10^5	No	3652	58*
S53Wos29	8	0	0	1088	No	1001	77*
S54Wos30	6	0	0	144	No	14400	90
S55Wos31	28	1	18	162	No	7560	69
S56Wos32	4	0	0	32	Yes	756	31
S57Wos33	16	1	32	288	Yes	43200	70
Starkey5	2	0	0	2	Yes	0	2
Starkey17	11	2	21	105	Yes	28800	69
Starkey23	6	2	512	4.05×10^5	No	28496	100
Starkey26	5	1	10	520	Yes	0	21
Starkey28	7	4	2.45×10^{12}	2.40×10^{25}	No	16786	275
Starkey29	7	2	441	7.83×10^5	No	16704	275
Starkey35	6	1	16	256	No	1840	41*
Starkey36	12	1	150	4.41×10^4	No	13911	171*
Starkey37	10	2	3600	5.51×10^7	No	19633	78*
Starkey41	3	0	0	9	Yes	0	8
Starkey55	4	2	576	1.39×10^6	No	7200	245
Starkey65	7	2	9408	2.68×10^6	No	2480	275
Starkey68	2	2	512	1.23×10^6	No	2754	275
Starkey75	10	3	1.23×10^6	6.87×10^{12}	No	1978	275
Starkey76	3	0	0	32	Yes	7	17
Starkey87	8	1	32	6272	No	4240	250
Starkey100	3	0	0	27	Yes	741	24
Starkey103	8	1	20	3332	No	26280	37
Starkey105	4	1	20	3332	Yes	43200	66
Starkey106	4	1	54	6.38×10^4	No	7920	66
Starkey108	10	1	324	1.73×10^7	No	12645	275
Starkey111	4	1	20	3332	Yes	27000	66
Starkey112	12	1	833	8.18×10^6	No	1517	275
Starkey115	5	1	176	6.54×10^6	No	8640	120
Starkey116	8	1	200	3.79×10^6	No	8640	130
Starkey118	12	1	1176	3.83×10^6	No	1220	250
Starkey121	7	1	176	6.54×10^6	No	7537	191

Table 2.1: The Great Theorem Prover vs Canonical Semantic Tree Generator.

these 46 theorems has an upper bound of at least 10000, and these are likely to be the hard theorems.

A modified version of The Great Theorem Prover was programmed to generate canonical semantic trees and it was given the Stickel Test Set for an exercise. The program found closed canonical semantic trees for 29 of the 84 theorems. The test was carried out during May of 1994 at McGill University's School of Computer Science using an IBM RS/6000[4] model 350. The program is approximately 8000 lines of C code, and is divided into two parts. The first part generates the canonical enumeration of atoms from the Herbrand base. The second part uses this enumeration of atoms to construct a semantic tree.

Columns 6-8 of Table 2.1 show the program's result on the Stickel Test Set. They specify whether a closed semantic tree was obtained for each theorem (Column 6), the execution time in seconds for the program to find a proof or to stop a search (Column 7), and the number of atoms checked before a proof was found or before the program terminated its search (Column 8). The program ceased searching if the number of resolvents on the path from the root to some node in the semantic tree became greater than the size of the clause database (in our case, the size of the database is 5000 clauses) or if 275 atoms have been used in building a semantic tree. These values may be increased depending on the size of the memory of the computer.

2.6 Obtaining Other Resolution-Refutation - Proofs From a Given Proof

Semantic trees can be used to obtain other proofs of unsatisfiability for a set of clauses from a given resolution-refutation proof for that set. This may be significant to artificial intelligence researchers, especially those who seek proofs of various categories. For instance, short proofs versus long proofs or cheap proofs versus expensive proofs (in terms of execution time and or computer's memory requirements).

[4]A trademark of International Business Machines Inc.

In order to obtain other resolution-refutation proofs from a given one, it is necessary to construct a closed semantic tree from the given proof. Once a closed semantic tree has been constructed, a simple manipulation of atoms labelling the branches of this semantic tree would provide a different closed semantic tree. By 'manipulation of atoms' we mean adding, deleting andor modifying some of these atoms. Other resolution-refutation proofs can be obtained from the newly constructed closed semantic trees. In what follows, Pascal-like pseudocode for this procedure is given.

```
Procedure Obtain_Proofs(P : Resolution_Proof);
Var
   X,Y : Herbrand_Base_Subset;
   T : Semantic_Tree;
   Q : Resolution_Proof;
Begin
      X = Resolved_Atom_Set(P);
      For every valid manipulation of atoms in the set X Do
      Begin
            Y = Set_Manipulation(X);
            T = Construct_Semantic_Tree(Y);
            Q = Extract_Proof(T);
            Print_Proof(Q);
      End
End
```

---

3 Improving Semantic Tree Generators

In the Stickel Test Set experiment of Chapter 2, we established that canonical semantic trees are poor devices for proving theorems in first-order predicate calculus. One of the primary objectives of this research is to improve the practicality of generating semantic trees for proofs of unsatisfiability. The hope is to improve the performance of semantic tree generators, in the sense that they can prove larger and more difficult theorems than can canonical semantic tree generators. The sequel suggests methods for achieving this objective.

3.1 Methods for Improving Semantic Tree - Generators

The performance of semantic tree generators can be improved in different ways, from which we have chosen to investigate three. In the first method, the Herbrand base is *filtered* by identification and elimination of useless and unnecessary atoms. In the second, atoms are selectively chosen from the Herbrand base by following certain control strategies in order to construct semantic trees. The effect of these strategies on the performance of a semantic tree generator

in producing proofs is examined using the Stickel Test Set. A discussion of their effectiveness on the outcome is presented in Section 5 of this chapter. The third and last method incorporates the use of an external human supervisor, as might be desirable, to assist the generator building the semantic trees.

Only theoretical aspects of the three proposed methods are addressed in this chapter. The actual implementation of these methods within a semantic tree generator is discussed in the next chapter. Henceforth, in this thesis a semantic tree generator will be called *improved* if it uses one of these three methods for generating its semantic trees. We close this chapter with a comparison made between a canonical semantic tree generator and an improved semantic tree generator from the viewpoint of the number of theorems which are proved from the Stickel Test Set by each method.

3.2 Method I: Filtering the Herbrand Base

It was noted earlier in Chapter 2 that the atoms of the Herbrand base determine the size and shape of the closed semantic trees of an unsatisfiable set of clauses. A difficulty that often arises when generating a semantic tree is that of using useless and unnecessary atoms from the Herbrand base. The use of such atoms usually forces duplicate subtrees to appear in the semantic tree. Consequently, one method for improving the performance of semantic tree generators is to *filter* the Herbrand base, keeping in mind that using useless and unnecessary atoms can be avoided at only a modest additional computational expense. The following two theorems justify allowing the elimination of useless and unnecessary atoms from the Herbrand base.

Theorem 3.1: *If φ is an unnecessary atom in the Herbrand base of an unsatisfiable set* S *of clauses such that a closed semantic tree for* S *is generated from a finite subset Λ of the Herbrand base with $\varphi \in \Lambda$, then another closed semantic tree can be generated from the subset $\Lambda' = \Lambda/\{\varphi\}$.*

Proof: Let φ be an unnecessary atom in the Herbrand base of an unsatisfiable

set S of clauses. Suppose a closed semantic tree Υ is generated from a finite subset Λ of the Herbrand base, with the atom φ labelling at least one of the branches of Υ at a certain depth. Since the order of atoms in Λ does not affect the fact that a closed semantic tree can be generated from the atoms of Λ, re-order the atoms in Λ such that φ and $\sim \varphi$ label the two branches descending from the root node of Υ as shown in Figure 3.1. Accordingly, re-construct the closed semantic tree Υ from the atoms of the modified Λ.

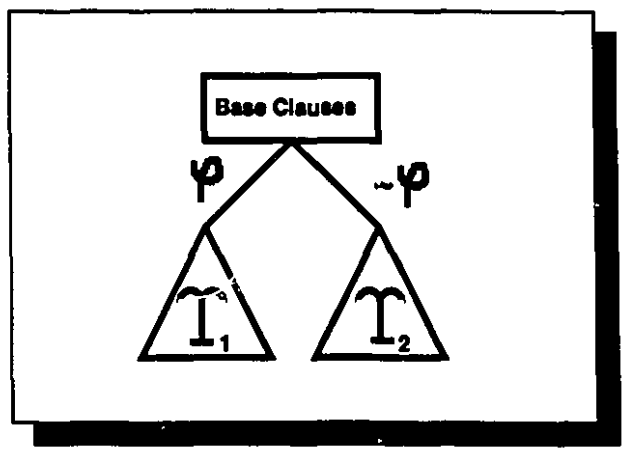

Figure 3.1: **A closed semantic tree Υ for S with two closed subtrees Υ_1 & Υ_2.**

Since Υ is a closed semantic tree, the two subtrees Υ_1 and Υ_2 of Υ must be closed. Moreover, since φ is an unnecessary atom, by definition either it or its complement but not both resolves with clauses in S. Without loss of generality, assume that φ does not resolve with any clause in S. Then, the closure of the left subtree Υ_1 does not depend on the atom φ; thus the branch labelled by φ can be eliminated. Moreover, the presence of the atom $\sim \varphi$ among those labelling the branches of the right subtree Υ_2 means that $\sim \varphi$ is an *uncomplemented literal*, which subsequently can be eliminated without affecting the completeness of this proving method. Therefore, another closed semantic tree for S can be generated from a finite subset Λ' of the Herbrand base which is equal to the subset Λ / $\{\varphi\}$. Q.E.D.

Theorem 3.2: *If φ is a useless atom in the Herbrand base of an unsatisfiable set* S *of clauses such that a closed semantic tree for* S *is generated from a finite subset* Λ *of the Herbrand base with* $\varphi \in \Lambda$ *, then another closed semantic tree can be generated from the subset* $\Lambda' = \Lambda/\{\varphi\}$.

Proof: Let φ be a useless atom in the Herbrand base of an unsatisfiable set S of clauses. Assume that there exists a closed semantic tree for S with the atom φ labelling at least one of its branches at a certain depth. Let Λ be a finite subset of the Herbrand base of S such that $\varphi \in \Lambda$ and that a closed semantic tree can be generated using the members of Λ. As in Theorem 3.1, re-order the atoms of Λ in such a way that φ and $\sim \varphi$ label the two branches descending from the root node of the closed semantic tree. But recall that φ is a useless atom, which means that neither it nor its complement resolve with any of the clauses of S. Therefore, the closure of the semantic tree does not depend on the atom φ or on its complement; thus the branches labelled by φ and $\sim\varphi$ can be eliminated. In other words, another closed semantic tree can be generated from a finite subset Λ' of the Herbrand base which is equal to the subset Λ / $\{\varphi\}$. Q.E.D.

The Filtering Procedure:

Given a Herbrand base (HB) of a set of clauses, the following is a Pascal-like pseudocode for a function **Filter** that accepts HB as input and returns a filtered version of it (FHB) as output:

```
Function Filter(HB : set_of_atoms) : set_of_atoms;
Var
   FHB : set_of_atoms;
   X : atom;
   EXIT : boolean;
Begin
      FHB := [];
```

```
    EXIT := false;
    While ( Not Empty(HB) And Not EXIT ) Do
    Begin
            X := Next_Enumerated_Atom(HB);
            if not Useless_Or_Unnecessary(X) then
              FHB := FHB + [X];
            if Generate_Closed_Semantic_Tree(FHB) then
              EXIT := true;
    End;
    Filter := FHB;
End;
```

The affect of filtering the canonical enumeration of atoms from the Herbrand base of a set of clauses on the performance of a semantic tree generator is best illustrated in the following example:

Example: Consider S_1 to be the following set of base clauses:

Axioms:

1. P(x)
2. ~P(a) | Q(f(a))

Negated Theorem:

3. ~Q(f(x))

The canonical enumeration of elements from the Herbrand universe of S_1 is: $HU(S_1) = \{ a, f(a), f(f(a)), f(f(f(a))), \ldots \}$.

The canonical enumeration of atoms from the Herbrand base of S_1 is: $HB(S_1) = \{ P(a), Q(a), P(f(a)), Q(f(a)), P(f(f(a))), Q(f(f(a))), P(f(f(f(a)))), \ldots \}$.

A closed semantic tree for S_1 is shown in Figure 3.2 (a). Both P(a) and ~P(a) resolve with some clauses in S. Therefore, the atom P(a) from the Herbrand base is neither useless nor unnecessary, whereas both Q(a) and ~Q(a) do not resolve with any clause in S. Thus, Q(a) is useless and can be elim-

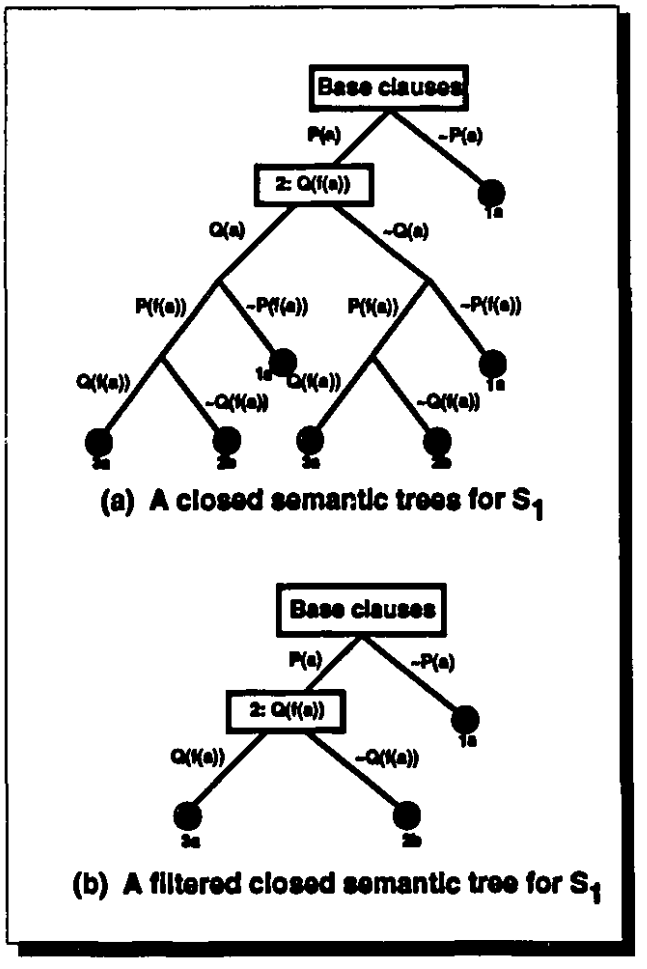

Figure 3.2: **Two closed semantic trees for the set S_1.**

inated from the Herbrand base. It can be seen in Figure 3.2 (a) how Q(a) caused redundant parts to appear in the closed semantic tree. Except for the atom Q(f(a)), all of the remaining atoms in the Herbrand base turned out to be unnecessary. Consequently, in this example, the *Filtered Herbrand Base* becomes $FHB(S_1) = \{ P(a), Q(f(a)) \}$, and a filtered closed semantic tree for the set S_1 is shown in Figure 3.2 (b).

Unfortunately, it is not often the case that filtering the Herbrand base reduces the number of atoms to be checked (by the generator) before a closed semantic tree is generated, as was true in the previous example. In order to investigate the effect of filtering the Herbrand base of a set of clauses on a semantic tree generator, we performed another experiment using the theorems

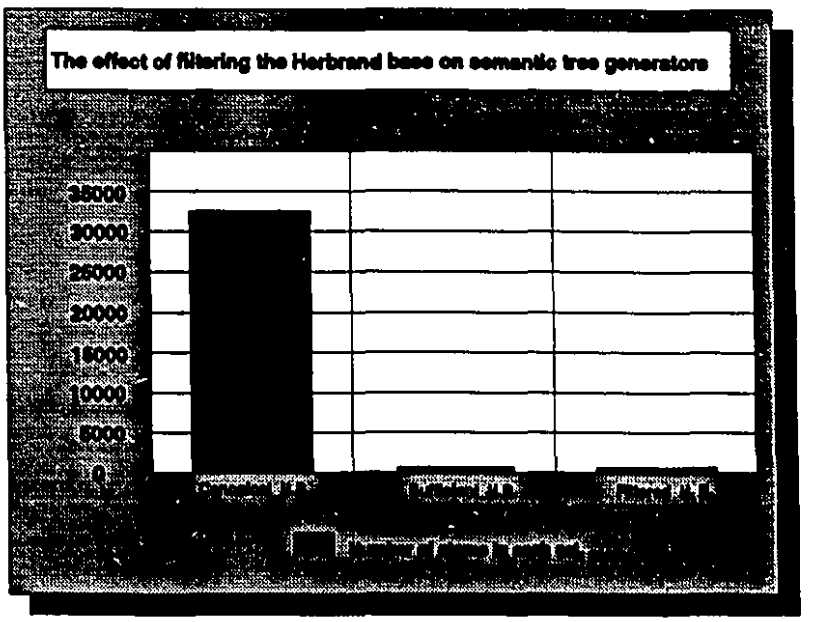

Figure 3.3: **Testing Method I on the Stickel Test Set.**

in the Stickel Test Set. This experiment compared the number of atoms filtered from the Herbrand base of each theorem with the number of atoms sufficient for proving that theorem. The graph in Figure 3.3 shows that these two quantities are approximately equal to each other and are relatively small compared to the number of Herbrand base atoms canonically enumerated up to and including all atoms sufficient for generating a closed semantic tree for each theorem in the Stickel Test Set.

3.3 Method II: Control Strategies for Semantic Tree Generators

A control strategy for a proof procedure searches for a refutation by attempting to grow a deduction tree by applying the inference rules selectively in a manner that sharply increases the effectiveness of the procedure [Luger1]. In general, a control strategy for a refutation system is said to be *complete* if its use results in a procedure that will eventually find a contradiction whenever one exists. However, for artificial intelligence applications such as mechanical theorem

proving, complete strategies are not as important as ones that find refutations efficiently.

Efficient control strategies for theorem-proving programs fall naturally into two classes: **choice strategies** and **edit strategies.** Choice strategies decide the order in which the deduction tree is to be generated. More importantly, they determine which part of the tree may be ignored (cut off) altogether. Edit strategies, on the other hand, eliminate trivial and superfluous deductions [Wos2]. If history is a good teacher, powerful edit strategies are of far greater value than are powerful choice strategies. The reason being that the former offer a greater reward by directly addressing the potential of combinatoric explosion and the latter address this obstacle only indirectly [Wos1].

Many efficient strategies for resolution-based procedures have been proposed and implemented, such as the *fewest literals*, the *set-of-support*, the *unit preference*, the *vine form* (also known as the *linear-input form*) and the *linear form* strategies [Nilsson2]. In what follows, we propose control strategies for semantic tree generators that are equivalent to those for resolution-dependent proof procedures.

A control strategy for a semantic tree generator searches for a refutation by attempting to grow a closed semantic tree from the atoms of the Herbrand base; the decision about the order of atoms that are to be used in generating the semantic tree is made irrevocably by the control strategy.

It is disconcerting that none of the research in tree searching techniques has yielded improved search strategies for theorem proving. Searching for paths in trees is not general enough to represent the searches needed in automatic theorem proving [Kowalski3]. Control strategies for semantic tree generators can be developed in analogy to existing strategies for the resolution-refutation methods which are well known. Our objective here is not to attempt to present an exhaustive set of strategies or even the most sophisticated techniques for proving theorems using semantic tree generators. Rather, we aim to bring to the attention of the research community the existence of such control strate-

gies, and to describe how they might be used in generating closed semantic trees for unsatisfiable sets of clauses.

3.3.1 The Fewest-Literals Strategy

Our basic motivation for proposing this strategy is to decrease the number of possible resolutions in play and hence to increase the efficiency of semantic tree generators which depend exponentially upon the size of the corresponding base clause set.

The fewest-literals strategy for resolution-refutation methods is one in which one-literal clauses are chosen first for resolution, followed by two-literals clauses, then three-literals clauses and so on. In short, preference is given to clauses with the fewest literals for resolution. The fewest-literals strategy for semantic tree generators is - more or less - similar to that for resolution methods. In this strategy, preference is given to those Herbrand base atoms which fail to satisfy one-literal clauses, followed by atoms that failed to satisfy two-literals clauses, then by atoms that fail to satisfy three-literals clauses, and so on.

The completeness of the fewest-literals strategy for semantic tree generators is guaranteed as it is for resolution methods. The reason for this is that all of the given clauses will eventually be used if enough resources are given to the semantic tree generators. By resources, we mean execution time and computer memory.

An algorithm for selectively choosing atoms from the Herbrand base using the fewest-literal strategy for semantic tree generators is given below:

1. Initialize the Herbrand base list (HB) to Nil.

2. If the atoms in the HB falsify every clause in S, then exit. Otherwise go to Step 3.

3. Select a ground literal L in S that is not already a member of HB. If no such ground literal is found, go to Step 4. Resolve L with the clauses in

S and add the resolvents r1, r2, ... to S. Call this set of resolvents R. If one of the resolvents in R also resolves with L, add these resolvents as well, enlarging R until no clause in R resolves with L. [For example for clause C = P(a,x) | P(a,y) | Q(x,y) in S and L = ~P(a,b), add r1 = (Ca,L) = P(a,x) | Q(b,x), r2 = (Cb,L) = P(a,x) | Q(b,x), r3 = (Cab,L) = Q(b,b).] If one of the resolvents in R is the null clause, then repeat the just-above described procedure using ~L in place of L. If no resolvent is added for ~L, then L is unnecessary. Otherwise, add L to HB. In either case go to Step 2.

4. Select a literal L in S that has arguments with only one variable. If no such literal is found, go to Step 5. Generate ground instances of L, with the variable replaced in each instance by a constant from amongst the constants appearing in S. Resolve each ground instance with the clauses in S and add these resolvents to S. Call this set of resolvents R2. If one of the resolvents in R2 also resolves with the ground instance, add these resolvents as well, enlarging R2 until no clause in R2 resolves with the ground instance. If one of the resolvents in R2 is the null clause, then repeat the just-above described procedure using the negation of the ground instance instead of the ground instance. If no resolvent is added for the negation of the ground instance, then the ground instance is unnecessary. Otherwise, add the ground instance to HB. In either case go to Step 2.

5. Select a literal L in S that has arguments with only two variables. If no such ground literal is found, go to Step 6. Generate ground instances of L, with the variables replaced in each instance by constants among those appearing in S. Resolve each ground instance with the clauses in S and add the resolvents to S. Call this set of resolvents R3. If one of the resolvents in R3 also resolves with the ground instance, add these resolvents as well, enlarging R3 until no clause in R3 resolves with the

ground instance. If one of the resolvents in R3 is the null clause, then repeat the just-above described procedure using the negation of the ground instance instead of the ground instance. If no resolvent is added for the negation of the ground instance, then the ground instance is unnecessary. Otherwise, add the ground instance to HB. In either case go to Step 2.

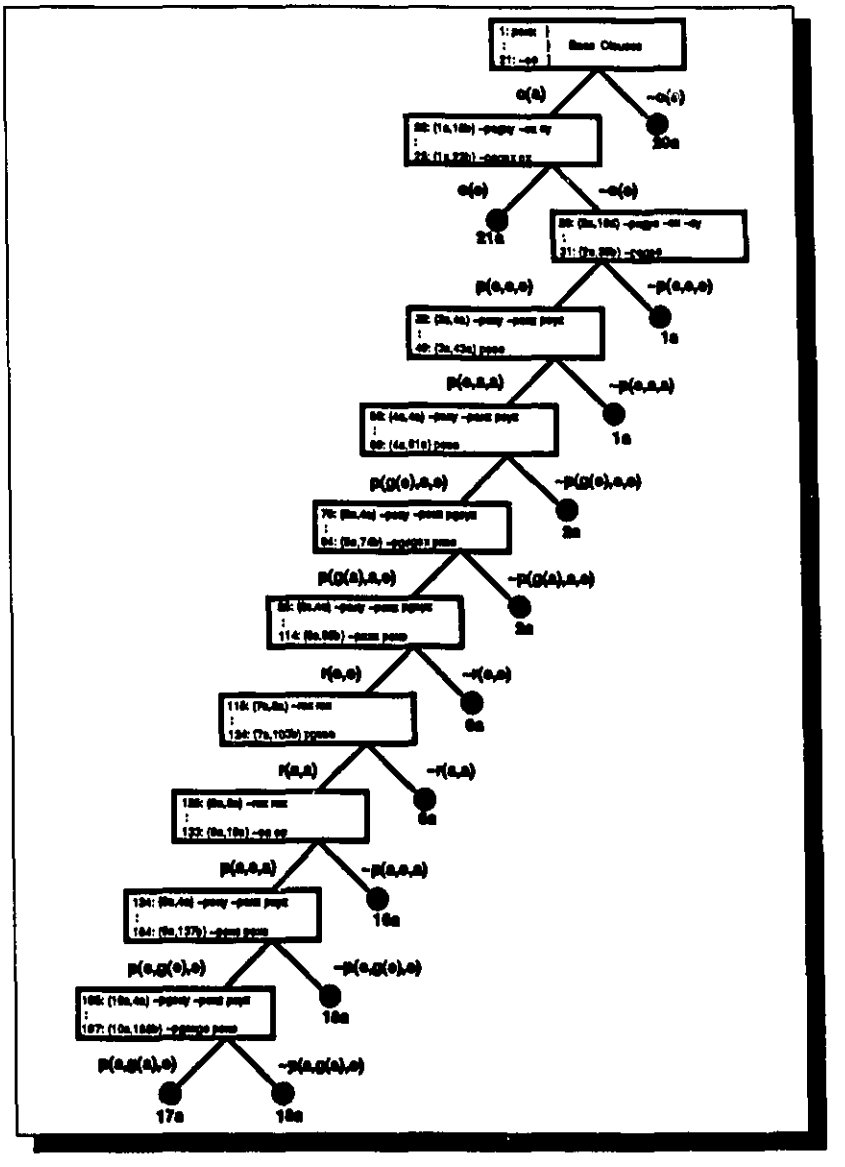

Figure 3.4: **A closed semantic tree for Wos12 generated using the fewest-literals strategy.**

6. Using the canonical enumeration of the Herbrand base, select an atom

that is not a member of HB. Resolve it with the clauses in S. Call this set of resolvents R4. If one of the resolvents in R4 also resolves with the atom, add these resolvents as well, enlarging R4 until no clause in R4 resolves with the atom. If one of the resolvents in R4 is the null clause, then repeat the just-above described procedure using the negation of the atom instead of the atom. If no resolvent is added for the negation of the atom, then the atom is unnecessary. Otherwise, add the atom to HB. In either case go to Step 2.

Example 1: Continuing to work with **Wos12**, o(a) is the first ground literal added to HB from clause 20 by Step 3. No closed semantic tree is generated using only o(a). o(e) is the second ground literal added to HB from clause 21 by Step 3. No closed semantic tree is generated using the two elements of HB and no more ground literals are available. Step 4 added p(e,e,e) and p(e,a,a) by substituting e for x and then a for x in clause 1. Also added are p(g(e),e,e) and p(g(a),a,e) by substituting e for x and then a for x in clause 2, likewise r(e,e) and r(a,a) by substituting e for x and then a for x in clause 6, and finally p(a,e,a) by substituting a for x in clause 16, and p(e,g(e),e) and p(a,g(a),e) by substituting e for x and then a for x in clause 18. A closed semantic tree is generated for Wos12 using the atoms in HB, and is shown in Figure 3.4.

3.3.2 The Set-of-Support Strategy

The set-of-support strategy is used to avoid generation of general lemmas, when a more focused search often produces a proof far more quickly [Wos4]. The set-of-support strategy for resolution-refutation methods is one in which at least one parent of each resolvent is selected from among the clauses of the negated conclusion (these are base clauses other than the given axioms) or from their descendants (i.e. the set-of-support). The *set-of-support Herbrand base* is one which includes atoms from the Herbrand base that resolve with, or have their complements resolve with, the clauses of the negated conclusion or the clauses of their descendants. The set-of-support Herbrand base will yield

a closed semantic tree for a set of clauses if the set is unsatisfiable. In this way, the set-of-support strategy for semantic tree generators is complete. A closed semantic tree for the set of clauses shown in Example 2 is generated from the atoms of the set-of-support Herbrand base (denoted by SOSHB(S_2)), and is shown in Figure 3.5.

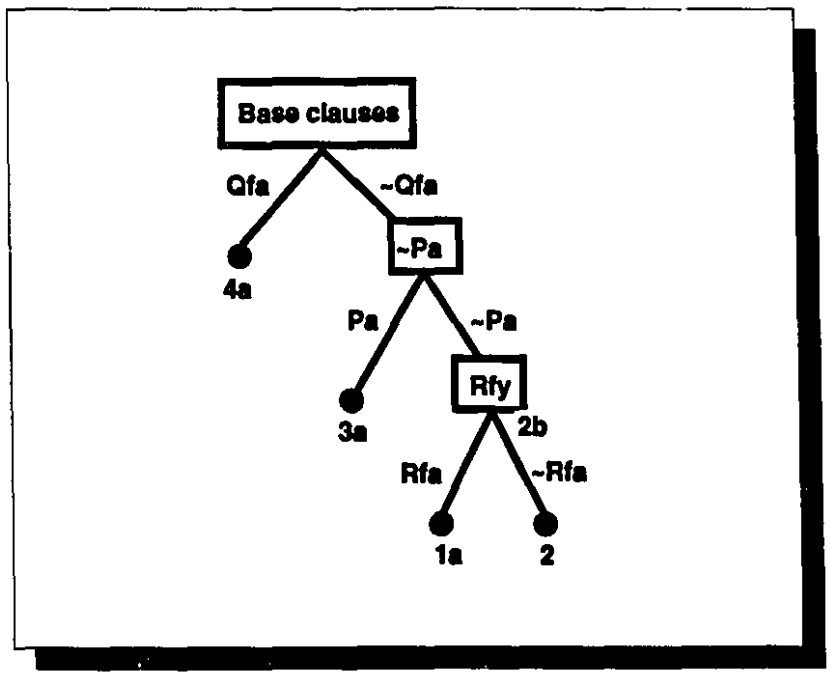

Figure 3.5: **A closed semantic tree for S_2 generated using the set-of-support strategy.**

Example 2: Let S_2 be the following set of clauses:

Axioms:

1. $\sim R(f(x))$
2. $P(x) \mid R(f(y))$
3. $\sim P(a) \mid Q(x)$

Negated Theorem:

4. $\sim Q(f(x))$

$HU(S_2) = \{ a, f(a), f(f(a)), \ldots \}$.

$HB(S_2) = \{ P(a), Q(a), R(a), P(f(a)), Q(f(a)), R(f(a)), P(f(f(a))), Q(f(f(a))), R(f(f(a))), P(f(f(f(a)))), Q(f(f(f(a)))), R(f(f(f(a)))), \ldots \}$.

SOSHB(S_2) = { Q(f(a)), P(a), R(f(a)), Q(f(f(a))), P(f(a)), R(f(f(a))), ... }.

3.3.3 The Unit-Preference Strategy

The unit-preference strategy for semantic tree generators is a modification of the previously presented strategy. Here, instead of selecting atoms from the Herbrand base that resolve with the clauses of the negated conclusion or their descendents, we select, by preference, those atoms from the Herbrand base that resolve with, or have their complement resolve with, single-literal clauses alone (i.e. unit clauses). Obviously, if the selection of atoms is restricted to those which resolve with unit clauses exclusively, then this strategy is not complete. If, for example, the clauses of the set S do not contain unit clauses, then this method stands defective.

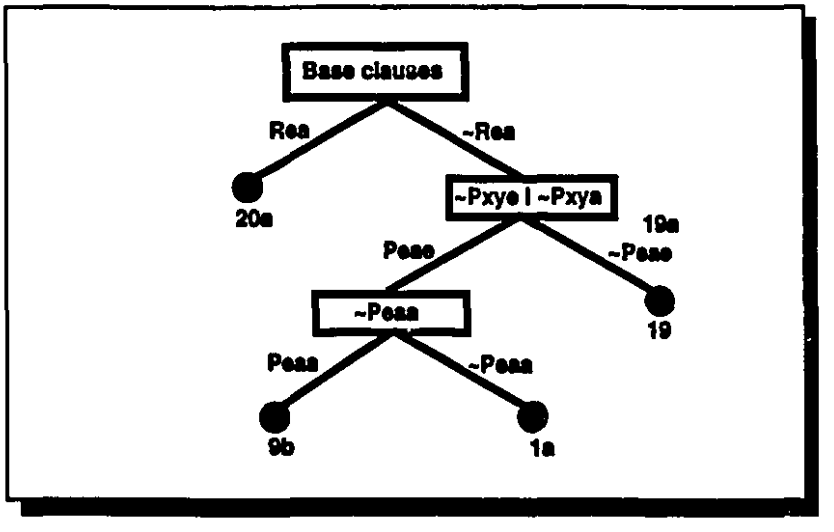

Figure 3.6: **A closed semantic tree for Wos3 generated using the unit-preference strategy.**

Example 3: Consider the theorem **S27Wos3.thm** (from the Stickel Test Set). The symbols e and a are constants.

Axioms:

01. P(e,x,x)
02. P(g(x),x,e)
03. ~P(x,y,z) | ~P(y,u,v) | ~P(z,u,w) | P(x,v,w)
04. P(x,y,f(x,y))
05. ~P(x,y,z) | ~P(y,u,v) | ~P(x,v,w) | P(z,u,w)
06. R(x,x)
07. ~R(x,y) | R(y,x)
08. ~R(y,x) | ~R(y,z) | R(x,z)
09. ~P(x,y,z) | ~P(x,y,u) | R(z,u)
10. ~P(z,u,x) | P(z,u,y) | ~R(x,y)
11. ~P(z,x,u) | P(z,y,u) | ~R(x,y)
12. ~P(x,z,u) | P(y,z,u) | ~R(x,y)
13. ~R(x,y) | R(f(z,x),f(z,y))
14. ~R(x,y) | R(f(x,z),f(y,z))
15. ~R(x,y) | R(g(x),g(y))
16. P(x,e,x)
17. P(x,g(x),e)
18. P(a,x,x)
19. P(x,a,x)

Negated Theorem:

20. ~R(e,a)

HU(Wos3) = { a, e, g(a), g(e), f(a,a), f(a,e), f(e,a), f(e,e), g(g(a)), g(g(e)), g(f(a,a)), ... }.

HB(Wos3) = { R(a,a), P(a,a,a), R(a,e), R(e,a), R(e,e), P(a,a,e), P(a,e,a), P(a,e,e), P(e,a,a), P(e,a,e), P(e,e,a), P(e,e,e), R(a,g(a)), R(a,g(e)), ... }.

UPHB(Wos3) = { R(e,a), R(a,e), P(e,a,e), P(a,a,a), P(e,a,a), ... }.

The closed semantic tree generated for Wos3 using this strategy is shown in Figure 3.6.

3.3.4 The Vine-Form Strategy

In the vine-form strategy (see Definition 1.3.31) for resolution-refutation methods, each resolvent has at least one of its parents belonging to the given axioms (these are base clauses other than the negated conclusion). The *vine Herbrand base* is, thus, defined to be those atoms from the Herbrand base that resolve with, or have their complement resolve with, the clauses of the given axioms. Like the resolution-refutation vine-form strategy, the vine-form strategy for semantic tree generators is not complete. In other words, the vine Herbrand base may not contain a sufficient number of atoms from the Herbrand base to generate a closed semantic tree. Example 4 demonstrates this fact.

Example 4: Let S_3 be the following set of clauses:

Axioms:

1. ~R(f(x))
2. P(x) | R(f(y))
3. ~P(a) | Q(x)

Negated Theorem:

4. ~Q(f(x)) | D(x)
5. ~D(x)

$HU(S_3)$ = { a, f(a), f(f(a)), f(f(f(a))), ... }.

$HB(S_3) = \{ D(a), P(a), Q(a), R(a), D(f(a)), P(f(a)), Q(f(a)), R(f(a)), D(f(f(a))),$
$P(f(f(a))), Q(f(f(a))), R(f(f(a))), D(f(f(f(a)))) P(f(f(f(a)))), Q(f(f(f(a)))), \ldots\}.$
$VHB(S_3) = \{ P(a), Q(a), P(f(a)), Q(f(a)), R(f(a)), P(f(f(a))), Q(f(f(a))),$
$R(f(f(a))), \ldots\}.$

It can be seen in the above example that the vine Herbrand base of the set S_3 does not contain any atom that resolves with the literal "Dx" in Clause 4 which, in turn, does not appear in the given axioms. Therefore, no closed semantic tree can be generated for S_3 using this strategy.

3.3.5 The Linear-Form Strategy

The linear-form strategy for resolution-refutation methods is one in which each resolvent has a parent that is either a member of the given axioms or is an ancestor of its other parent. Similarly, the *linear Herbrand base* is one which contains atoms from the Herbrand base that resolve with, or have their complements resolve with, the given axioms or their descendents. The vine-form strategy for semantic tree generators is a special case of the linear-form strategy presented here. One may note that the difference between the vine and the linear Herbrand base consists of those atoms in the linear Herbrand base that resolve with, or have their complements resolve with, clauses which descend from the given axioms. However, unlike the vine-form strategy, the linear-form strategy for semantic tree generators is complete as is true for resolution-refutation methods, and the atoms in the linear Herbrand base may yield a closed semantic tree for an unsatisfiable set of clauses. Example 5 illustrates the mechanization of this strategy.

Example 5: Let S_4 be the following set of clauses:

Axioms:

1. $\sim R(f(x))$
2. $P(x) \mid R(f(y))$
3. $\sim P(a) \mid Q(x)$

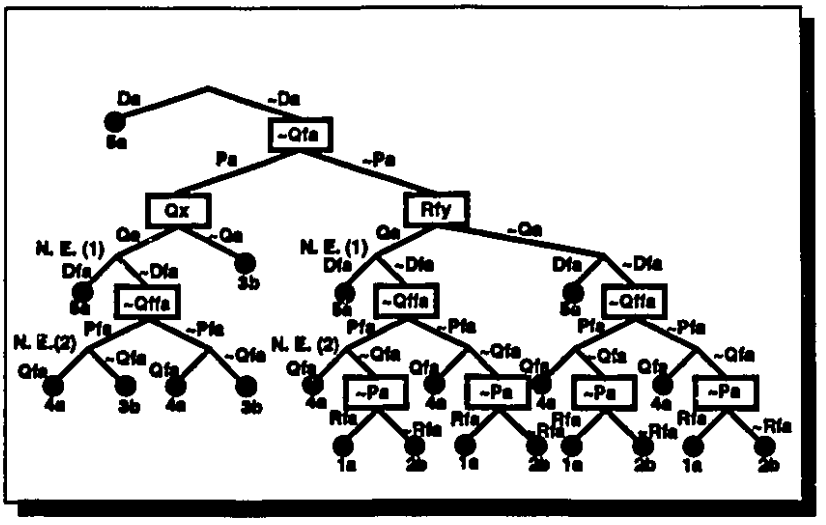

Figure 3.7: **A closed semantic tree for S_4 generated using the linear-form strategy.**

Negated Theorem:

4. $\sim Q(f(x)) \mid D(x)$

5. $\sim D(x)$

$HU(S_4) = \{ a, f(a), f(f(a)), f(f(f(a))), \ldots \}$.

$HB(S_4) = \{ D(a), P(a), Q(a), R(a), D(f(a)), P(f(a)), Q(f(a)), R(f(a)), D(f(f(a))),$
$P(f(f(a))), Q(f(f(a))), R(f(f(a))), D(f(f(f(a)))), P(f(f(f(a)))), Q(f(f(f(a)))), \ldots \}$.

$LFHB(S_4) = \{ D(a), P(a), Q(a), D(f(a)), P(f(a)), Q(f(a)), R(f(a)), D(f(f(a))),$
$P(f(f(a))), Q(f(f(a))), R(f(f(a))), \ldots \}$.

A closed semantic tree for S_4 is generated from the linear Herbrand base, and is shown if Figure 3.7. The closed semantic tree given in Figure 3.8 is generated using the same linear Herbrand base set with both useless and unnecessary atoms filtered.

3.3.6 Other Strategies

We have proposed to this point several control strategies for semantic tree generators. We believe that these strategies can be quite effective in controlling the generation of closed semantic trees of unsatisfiable sets of clauses. But, these strategies are not the only ones known to us. Other control strategies

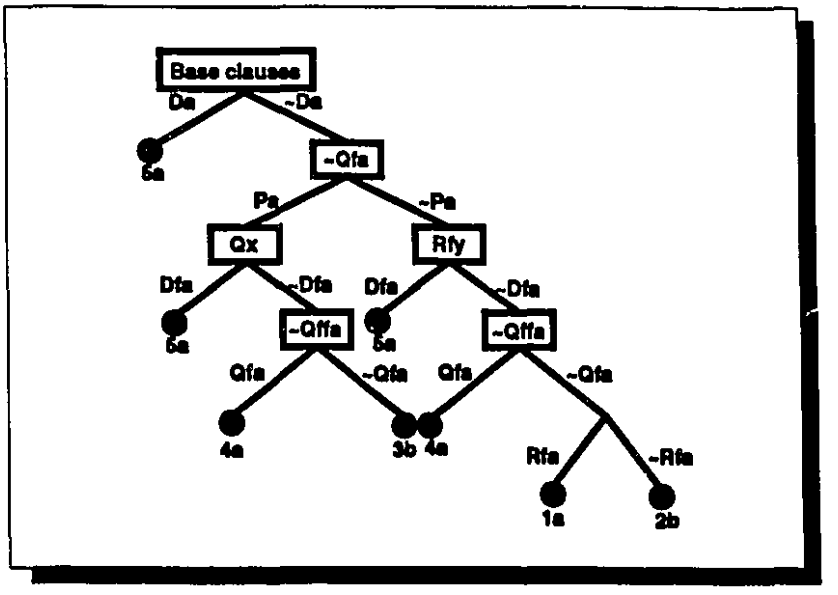

Figure 3.8: **A closed semantic tree for S_4 generated using the filtered linear-form strategy.**

can be obtained from various combinations of the control strategies presented earlier. For instance, a hybrid of the set-of-support strategy and the unit-preference strategy was recommended by Wos for resolution proof procedures [Wos2]. A similar combination can be used for semantic tree generators. It is also possible to combine the set-of-support strategy with the vine-form strategy as suggested in [Nilsson2].

Search heuristics may also be included into the design of semantic tree generators. This can be done by employing a left-right ordering of atoms in the Herbrand base. Many techniques are available in the literature to govern the ordering of atoms in the Herbrand base, such as the *best latent semantic clash preference strategy* [Robinson3], the *merit-ordering heuristic* [Robinson3] and the *diagonal search strategy* [Kowalski2].

3.4 Comparative Study

An improved semantic tree generator was implemented to measure and analyze the effect of the control strategies on the generation of closed semantic

trees. The fewest-literals strategy *(FLS)*, the set-of-support strategy *(SOS)*, the linear-form strategy *(LF)*, a hybrid of FLS and SOS and a hybrid of FLS and LF are all supported within this generator. We tested this semantic tree generator on the Stickel Test Set and compared the result with it for the canonical semantic tree generator which was obtained in Chapter 2. The result of this experiment is shown in the table presented in Appendix A at the end of this thesis.

The first column of the table shown in the appendix indicates the name of the theorem. The remaining columns in the table are divided into six groups of three columns each with an extra column at the end of the table. The first group corresponds to the canonical semantic trees, and the next five groups correspond to the five just-above mentioned control strategies. Each group is titled with the name of the strategy by which theorems were attempted. The first column in every group reveals whether a closed semantic tree was obtained for each theorem. The second column shows the number of atoms checked before a proof was found or before the program stopped searching, which would occur when the number of resolvents along the path to some node in the semantic tree became greater than the size of the clause database (in our case, the size of the database is 5000 clauses) or if 275 atoms have been used in building a semantic tree. These values may be increased depending on the size of the memory of the computer. The third column in each group shows the execution time for the program to find a proof or to stop searching. The last column in the table indicates the best among the six strategies for proving the Stickel theorem in question.

Figure 3.9 summarizes the outcome of this experiment, showing the ratio of number of theorems proved by each of the six control strategies.

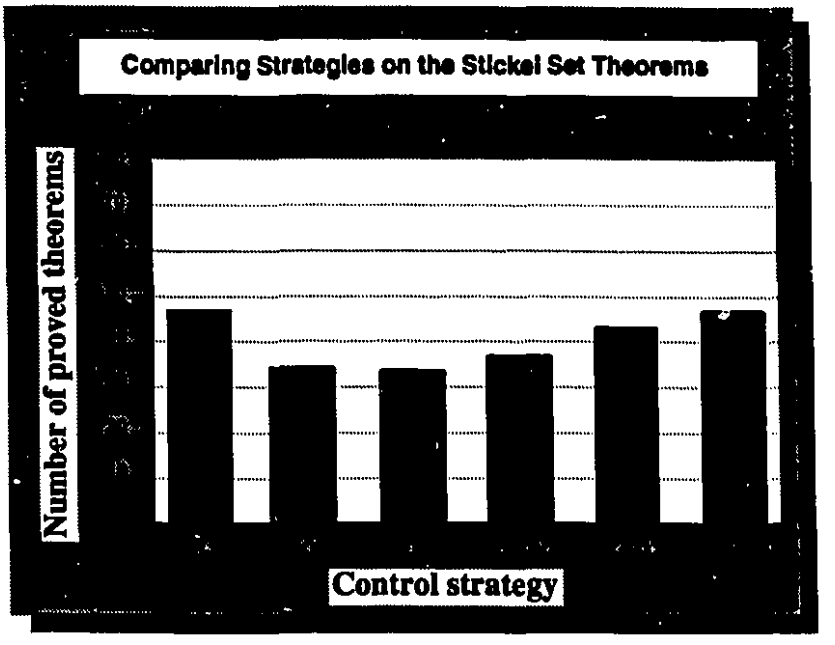

Figure 3.9: **Comparing the effect of control strategies on semantic tree generators.**

3.5 Method III: Advice-taking and Knowledge Programming within Semantic Tree Generators

Early artificial intelligence work studied adaptive learning schemes that could adjust control parameters to correlate the machine's output with a desired standard. Over time, artificial intelligence researchers moved increasingly toward a belief that acceptance of human advice and knowledge about the task should be integrated to intelligent behavior. This was seen necessary to reduce the amount of unexpected results produced by the intelligent system. Iterative refinements generally cause programs to become progressively more obtuse in their control structures. This analysis suggests an alternative paradigm for the programming and iterative refinement of intelligent systems. This paradigm views the programming problem primarily as one of translating expert advice into an operational program, and the iterative improvement problem as one of diagnosing program behavior to modify those elements that produce

undesirable behaviors [Klahr1].

This proposed scheme emphasizes the problems of understanding high-level advice, converting it into effective behavior, and, inevitably, changing the knowledge and reiterating the cycle. These problems are referred to as *knowledge acquisition, knowledge programming,* and *knowledge refinement,* respectively [5]

In the remainder of this section, we explain the integration of these problems into the task of generating semantic trees from the atoms of the Herbrand base. While generating the semantic trees, we allow for an external human supervisor and/in an interactive session to control the order of atoms in the Herbrand base. In other words, we allow the supervisor to alter the Herbrand base in such a way as to revive the generation process.

Four operations are required in order for a human supervisor to fully control the construction of semantic trees. These operations are used to manipulate the order of atoms in the Herbrand base, and are listed below:

1. Add an element to the Herbrand base.

2. Delete an element from the Herbrand base.

3. Exchange the position of two elements in the Herbrand base.

4. Insert an element at a random position in the Herbrand base.

With knowledge of the domain, the supervisor can control the direction(s) in which the semantic tree grows in such a way that closed semantic trees are generated as fast as possible. This feature can be most effective when other proofs of unsatisfiability from a given one are being sought (refer to Section 2.6).

[5] Knowledge acquisition, in our paradigm, refers to the transfer of expertise from a human expert to a machine. The machine *acquires* a person's knowledge through interactive sessions. When a machine extends its initial knowledge by various learning methods, we refer to this as *knowledge refinement.* Different researchers might apply the term *knowledge acquisition* to varying aspects of these processes.

4 The AISTG: An Improved Semantic Tree Generator

In Chapter 3, we suggested theoretical methods for improving the performance of semantic tree generators as mechanical theorem provers. In this chapter, we put these methods into practice by implementing a semantic tree generator embodying these methods and by testing it on the Stickel Test Set. The AISTG is an improved semantic tree generator, developed specifically for the purpose of improving the practicality of generating semantic trees for proofs of unsatisfiability. It also plays the role of a first-order predicate theorem prover.

What we will describe here is essentially a project towards the goal of developing a flexible yet practical theorem prover. Noting that many powerful and versatile theorem-proving programs exist, one naturally wonders about the ease of using such a program and which program is recommended. The program we recommend is the one which plays a vital role in our research: its name is AISTG. This program uses binary resolution for its inference mechanism and includes factoring as one of its abilities. Factoring is used in the AISTG only for extracting resolution-refutation proofs from generated closed semantic trees (refer to the algorithm presented earlier in Section 2.4). The

AISTG is written in C, and runs on any computer hosting *Unix*[6] as its operating system. Restricting the running environment to Unix does not impose a limit on the usage of the AISTG program. Minor modifications to the configuration file of the program make it portable to other computers hosting different operating systems.

The AISTG program can prove any theorem expressed in propositional or first-order predicate calculus. Yet, it may take a long time to prove some of the first-order predicate theorems. This is due to the vast number of atoms in the Herbrand base that need to be checked before proofs are found for these theorems (i.e. before closed semantic trees are generated).

From the standpoint of control strategies, the AISTG is distinguished from other theorem-proving programs. For the purpose of generating semantic trees the AISTG uses two simultaneous control strategies. The first one is for selecting atoms from the Herbrand base of the given clauses. The second control strategy is for generating a semantic tree from those atoms. The AISTG permanently uses depth-first iteratively-deepening [Almulla2, Letz1] for generating a semantic tree from the atoms of the Herbrand base. On the other hand, the control strategy for selecting atoms from the Herbrand base has to be chosen by the user prior to generating the semantic tree.

4.1 General Description of the AISTG Program

The AISTG program attempts to prove the unsatisfiability of some clauses by generating a semantic tree for them. If the clauses are unsatisfiable, the program will eventually generate a finite closed semantic tree. Otherwise, the semantic tree is infinite, which means that the program continues attempting to generate a closed semantic tree until it runs out of resources. Once a closed semantic tree has been generated, upon request by the user an algorithm for extracting a resolution-refutation proof from the closed semantic tree is invoked

[6] A trademark of AT&T Bell laboratories

and a proof is eventually printed. The algorithm for extracting resolution-refutation proofs from closed semantic trees was presented in Chapter 2.

The AISTG can prove theorems within various mathematical domains, such as *plane geometry, set theory, number theory*, and *algebraic structures including rings, fields, and groups.* The theorem-proving ability of the AISTG accommodates not only mathematical domains, but also puzzles which are usually found in *artificial intelligence, syllogisms*, and *cognitive science.* For example, the lion and the unicorn problem, the knights and knaves problem, and the monkey and the banana problem [Newborn2, Loveland2].

Our improved semantic tree generator is designed to be of interest to mathematicians, logicians as well as artificial intelligence researchers (specially those in automated theorem proving). The structure of the program is divided into eight modules and can be extended. The following is a description of these modules.

1_ **Stg.c:** Among all modules of the AISTG, the Stg.c is the most important one. It contains the main program which activates the procedures and functions constituting the engine of this theorem prover. First, the module reads (in text format) and converts (into binary format) the base clauses; asks the user to choose a control strategy for selecting atoms from the Herbrand base. Second, the program starts selecting atoms according to the control strategy chosen by the user. If the base clauses are unsatisfiable, the AITSG will eventually generate a closed semantic tree. Otherwise, the program keeps executing until one of the following thresholds is reached: the maximum allowed execution time, the memory available for the program or the maximum number of iterations allowed.

2_ **Compile.c:** This module receives as input the binary format of the base clauses that was generated by Stg.c, and it constructs the internal memory representation of these clauses according to the data structures of the AISTG program. Once the base clauses have been represented in

memory, the program starts selecting atoms from the Herbrand base according to the chosen control strategy.

3_ **Search.c:** The order by which the atoms are selected from the Herbrand base for generating a semantic tree depends on the control strategy. The program places a menu of five control strategies from which the user is to choose. This facility of the AISTG allows the users to take advantage of their personal experience in choosing an appropriate control strategy based on the theorem in question. Procedures for implementing the five control strategies of the AISTG can be found in this module.

4_ **Infer.c:** The AISTG program uses **binary resolution** as an inference rule for generating closed semantic trees of unsatisfiable sets of clauses. Therefore, the completeness of the AISTG theorem-proving method is guaranteed, since the resolution principle of Robinson is complete. The completeness of the resolution principle was proved in [Hsiang1, Robinson3, Robinson5, Slagle3]. A proof for the completeness of the AISTG proving method was outlined in Chapter 3. All procedures and functions concerning the inference rule of the AISTG can be found in this module.

5_ **Gen.c:** This module contains procedures and functions required for generating all binary resolvents of a given pair of clauses. Duplicating a clause is another task performed by the procedures of this module, however such clauses are needed by the AISTG program to cover cases where factoring literals of newly generated resolvents is required.

6_ **Unify.c:** Unification is a vital operation in the resolution process. Since we are using binary resolution as the inference rule of the AISTG, the resolved away literals in the parent clauses must be unified. Procedures for unifying the resolved away literals can be found in this module.

7_ **Dump.c:** This module is responsible for the "output" of the AISTG program. Upon the generation of a closed semantic tree for an unsatisfiable

set of clauses, procedures in this model are activated to print the content of the failure nodes in the closed semantic tree, as well as to print the resolution-refutation proof extracted from the tree (if requested by the user). It is usually necessary for a theorem-proving program to be able to reconstruct detailed information from any proof it generates. Procedures in Dump.c are called if the user requests to dump the content of the clause database during the generation of the semantic tree. This provides for the possibility of using the AISTG to prove theorems in non-mathematical domains such as information retrieval.

8_ **Hash.c:** Searching for duplicate resolvents, and thus for duplicate proofs, often wastes a great deal of execution time in theorem-proving programs. Wos [Wos3] wonders *what strategy can be employed to deter a reasoning program from deducing a clause already retained, or from deducing a clause that is a proper instance of a clause already retained?* Spencer addressed this problem in his paper *avoiding duplicate proofs* [Spencer1]. Newborn solved the same problem by assigning hash codes to the literals and the clauses of The Great Theorem Prover [Newborn1]. The latter approach was adopted to solve this problem in the AISTG program. The literal and the clause are assigned unique hash codes. These codes make the search for duplicate resolvents significantly efficient. Procedures for hashing and checking for duplicate literals and clauses can be found in this module.

In addition to the eight modules described above, the AISTG program includes two files. One contains the declaration of the data structures used by the program. This file is called **Clause.h**. The other file identifies all constants of the program and initializes their values. This file is called **Const.h**.

4.2 Flow of Control in the AISTG Program

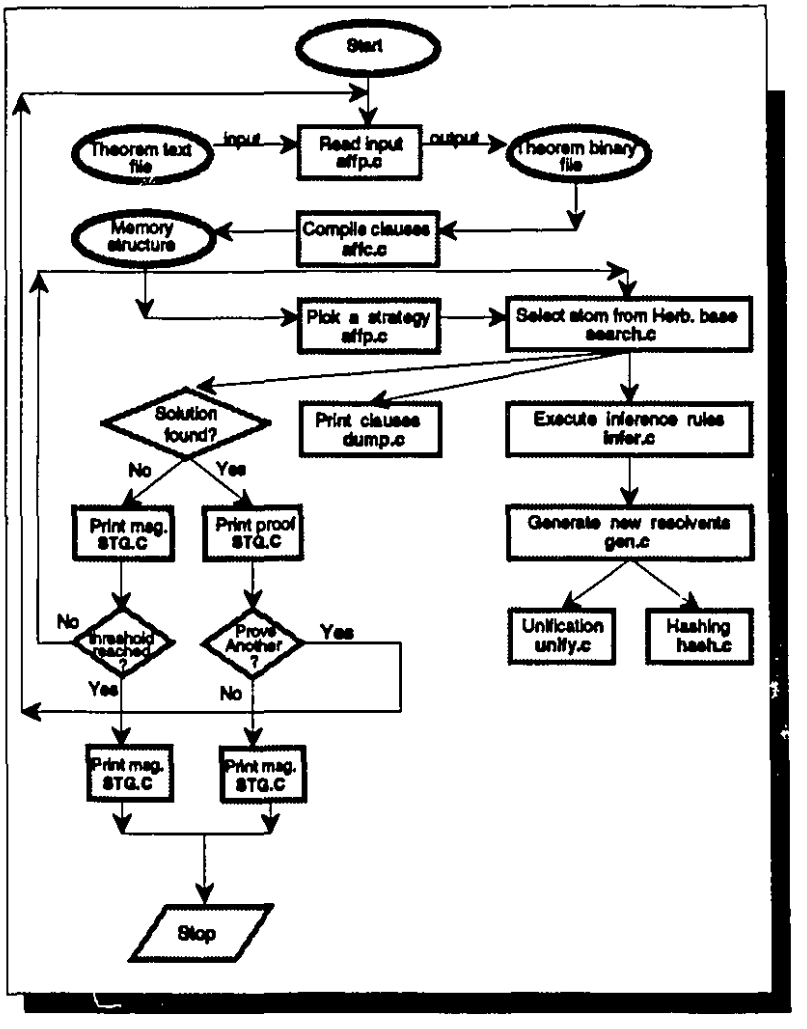

Figure 4.1: **Flow of control in the AISTG program.**

In this section, we trace the flow of control of the AISTG. The module Stg.c receives as input a text file containing the base clauses and creates an output file containing the base clauses in binary format. The binary file becomes input to the module Compile.c, which creates a representation of the base clauses in internal memory. The flow of control then goes back to the module Stg.c, where the user must choose a control strategy for selecting atoms from the Herbrand base. These atoms are, in turn, to be used to generate the semantic tree.

Once a control strategy has been chosen by the user, the flow transfers

to Search.c where the procedures for implementing the control strategies are found. Upon a selection of an atom from the Herbrand base, the inference rule procedure in Infer.c is called to generate all resolvents of the atom with the clauses in the clause database. In turn, the inference rule procedure calls procedures in Gen.c and Dump.c both to generate and to print binary resolvents, as well as to update the clause database. After updating the clause database, the inference rule procedure checks to see if a closed semantic tree has been generated, in which case a proof is found. Otherwise, another atom is selected from the Herbrand base by the control strategy. A diagram for the fundamental flow of control in the AISTG program is shown in Figure 4.1, along with the general layouts of the program.

4.3 Using the AISTG Program

Important design considerations in the AISTG program are performance, portability, compactness and simplicity of code. The program runs on computers hosting Unix as their operating systems. However, small modifications to the file Stg.c make the program portable to other computers hosting DOS[7] or Macintosh environments. For running the AISTG on Unix, one needs the ten files specified in Section 4.1 as well as the file "MAKEFILE" which compiles and links these ten files. To create an executable version of the AISTG program, at the computer prompt the user should type:

```
Unix:> make
```

An executable version of the AISTG (that is called **hprove**) is thus created. The AISTG program is now ready for use. To prove a theorem, the user should type the word:

```
Unix:> hprove
```

[7]Disk Operating System

the AISTG responds with the following question:

```
Enter the name of the theorem file (type '?'  for help):
```

At this point the user should enter the name of a text file where the theorem is saved. The AISTG reads the input file, creates the data structures for the base clauses in memory, and displays the clauses on the screen. After setting up the memory structures, the program prompts the user with a menu to choose one of the five control strategies available in the AISTG. The control strategy selection menu is shown in Figure 4.2.

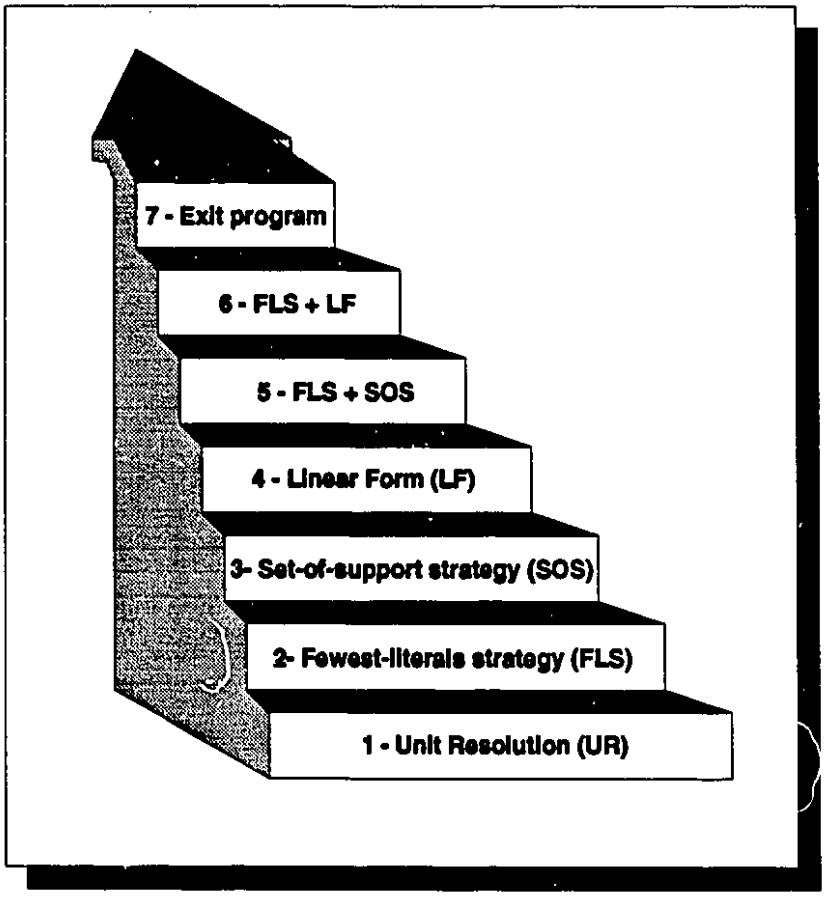

Figure 4.2: **The control strategy menu.**

Once a choice has been made by the user, the program starts generating a semantic tree from the atoms selectively picked from the Herbrand base by the control strategy. One useful feature in the AISTG program is the feedback it

gives to the user about the status of the prover and the state of the proof. The program shows the semantic tree on the screen as it is being created. Hence, the user is not kept in suspense until a proof is found.

4.4 Interactiveness of the AISTG Program

> Apart from the theoretical and practical limitations of automated theorem-proving systems, what of the more pragmatic issues such as: will the proofs derived by a machine provide "insight" to a human user, or are they essentially "plodding" and "tedious", essentially "trying all possible combinations" and providing nothing more than an answer of "true"? These are, of course, subject issues, but even so they are of importance to designers building systems for practical applications. With regard to elegance, it may be said that there have been proofs performed by computers of well-known theorems that have both generalized the known results and been more "elegant" than the human proofs. In general, however, the practicability of a system is perhaps more dependent upon the "naturalness" of its modes of inference and its interactiveness rather than the subtlety of its reasoning.
>
> [Wos3]

As previously noted, the AISTG program uses depth-first iterative-deepening as a permanent strategy for generating semantic trees of given sets of clauses. Each iteration extends the depth of the semantic tree by one level. Upon completion of each iteration, the program asks the user to press any key on the keyboard in order to continue building the semantic tree. If the user at that point responds by pressing the character "+", the AISTG program asks the user the following question:

```
Modify The Herbrand Base?  (Y/N):
```

Pressing the character "y" or "Y" on the keyboard invokes an interactive menu. Through this menu, the user can manipulate the order of atoms selected to that point. Then, it asks the user through the menu if s(he) wants to add a new atom to the end of the list, to delete a particular atom from the list, to exchange the position of two atoms, or to add a new atom at a specific position in the list to be chosen by the user. These four options give the

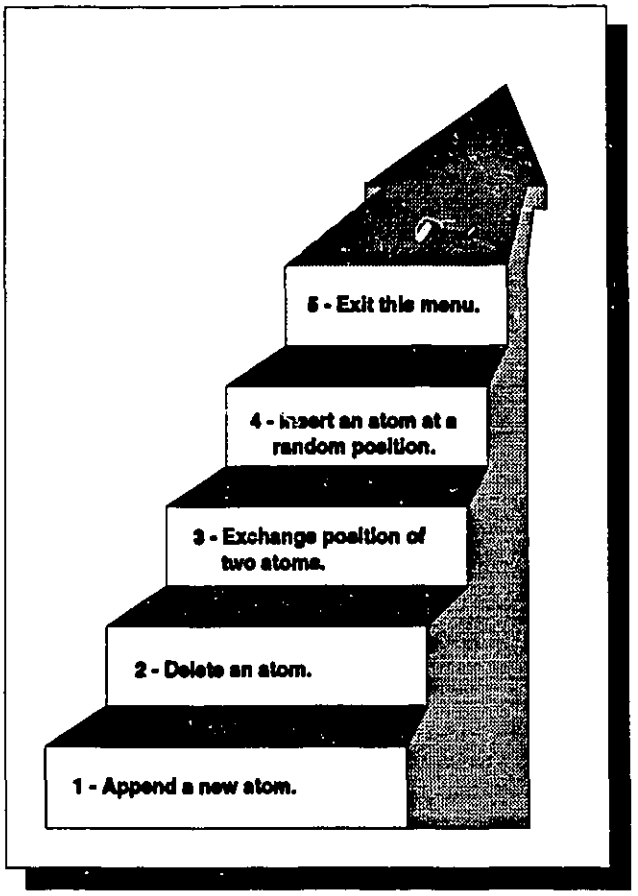

Figure 4.3: **The Herbrand base manipulation menu.**

user total control over the list of atoms to be used in generating a semantic tree for the base clauses. The AISTG program re-displays the Herbrand base manipulation menu following the execution of each choice made by the user with the exception of the fifth and last choice in the menu, which instructs the program to re-generate a semantic tree for the base clauses according to the newly made changes in the enumeration of atoms from the Herbrand base. The Herbrand base manipulation menu is shown in Figure 4.3.

Sample runs of the AISTG program on theorems arbitrarily chosen from the Stickel Test Set are exhibited in Appendix C. Similar sample runs for a canonical semantic tree generator are exhibited in Appendix B. The actual code of the AISTG, along with the executable version of the program can be obtained by sending an electronic mail to the author's address: *almulla@opus.cs.mcgill.ca*

or by contacting him via the School of Computer Science at McGill University.

4.5 Capabilities and Limitations of the AISTG Program

The AISTG has the potential for proving some theorems from the Stickel Test Set which seemed difficult to prove by using canonical semantic tree generators. This is not always true; in Chapter 5 we show theorems for which the proofs obtained using the canonical semantic tree generator were faster than those obtained using the AISTG. Nonetheless, AISTG has difficulty in proving other theorems from the Stickel Test Set. This section outlines the capabilities and limitations of the AISTG program.

4.5.1 Capabilities

Although some parameters appeared in the Const.h file to have maximal values, these parameters are computer-memory dependent and can easily be changed to accommodate larger quantities (if desired). Hence, these parameters can be considered as features of the AISTG program. Examples of these parameters are:

1. Total number of base clauses in the theorem file.
2. Total number of literals (predicates) in a clause.
3. Total number of variables in a clause.
4. Total number of functions in a predicate.
5. Total number of constants in a predicate.
6. Total number of arguments in a function.
7. Total number of atoms in the Herbrand base.
8. Total number of iterations performed before the AISTG stops the search for a proof.

9. Total number of clauses in the clause database (including the base clauses).

4.5.2 Limitations

The AISTG does not attempt to find a proof for a theorem expressed as a set of wffs (see Definition 1.3.4). These wffs should, first, be converted to clause form, then be fed to the AISTG as base clauses. An eight-step algorithm for converting wffs to base clauses can be found in [Newborn1, Nilsson2]. The base clauses, in turn, must take the form of some axioms and a negated conclusion. The user must negate the conclusion before adding it to the base clauses. Since the AISTG is a descendent of The Great theorem prover [Newborn1], one can use the COMPILE package of The Great Theorem Prover to convert wffs into clause form for the AISTG.

4.6 AISTG vs The Great Theorem Prover

The performance of the AISTG on the Stickel Test Set was analyzed previously in Chapter 3, and a summary of the result was given in Figure 3.9. In Table 4.1 the performance the AISTG program using the fewest-literals strategy[8] is compared with that of The Great Theorem Prover [Newborn2] using the IBM RS/6000 machine.

Table 4.1 presents the name of each theorem in the Stickel Test Set (Column 1), Columns 2-4 of the Table show the result of proving the Stickel Test Set using The Great Theorem Prover. They specify whether a proof was obtained for each theorem (Column 2), the execution time in seconds for the program to find a proof (Column 3), and the length of the proof (Column 4). Columns 5-7 of Table 4.1 show the result of proving the Stickel Test Set using the AISTG. They specify whether a closed semantic tree was obtained for each theorem (Column 5), the execution time in seconds for the program to find a proof or to stop searching (Column 6), and the number of atoms checked before a proof

[8] Refer to the algorithm given in Section 3.3.1

[9] The asterisk character in this column signals an overflow in the number of resolvents generated.

Theorem Name	The Great Theorem Prover			AISTG		
	Proven	Time (in sec.)	Length	Proven	Time (in sec.)	Length
S01burst	Yes	1	12	Yes	8	45
S02short	Yes	0	6	Yes	0	11
S03prime	Yes	1	15	Yes	217	31
S04haspa1	Yes	0	8	No	81	361
S05haspa2	Yes	0	13	No	208	381
S06ances	Yes	0	6	Yes	1	6
S07NUM1	Yes	0	6	Yes	2	17
S08group1	Yes	0	4	No	2400	96
S09group2	Yes	1	10	Yes	40	26
S10ew1	Yes	0	6	Yes	0	5
S11ew2	Yes	0	5	Yes	1	3
S12ew3	Yes	0	10	Yes	1	5
S13rob1	Yes	0	6	Yes	0	10
S14rob2	Yes	1	10	Yes	32	25
S15michie	Yes	0	4	Yes	19440	182
S16qw	Yes	0	8	Yes	0	3
S17mqw	Yes	0	5	Yes	0	3
S18DBABHP	Yes	0	11	No	18077	181
S19APABHP	Yes	69	15	No	567	389
S20fleisig1	Yes	3	11	Yes	94	30
S21fleisig2	Yes	2	11	Yes	32	34
S22fleisig3	Yes	1	20	No	4250	212
S23fleisig4	Yes	7	18	No	7960	74
S24fleisig5	Yes	3	18	No	8760	74
S25Wos1	Yes	2	7	No	4513	226
S26Wos2	Yes	1	6	No	7414	132
S27Wos3	Yes	0	5	Yes	0	13
S28Wos4	Yes	3	15	Yes	1500	160
S29Wos5	Yes	1	7	No	5049	174
S30Wos6	Yes	0	9	No	9500	130
S31Wos7	Yes	1	8	Yes	118	46
S32Wos8	Yes	0	8	No	8265	213
S33Wos9	Yes	0	7	No	7206	149
S34Wos10	Yes	2	10	Yes	266	34
S35Wos11	Yes	1	9	No	12631	141
S36Wos12	Yes	0	4	Yes	0	11
S37Wos13	Yes	1	6	Yes	1	16
S38Wos14	Yes	1	7	Yes	3	23
S39Wos15	Yes	16	16	No	4080	212
S40Wos16	Yes	1	7	Yes	397	100
S41Wos17	Yes	1	9	Yes	18000	135
S42Wos18	Yes	1	5	Yes	0	8

Table 4.1: **The Great Theorem Prover vs AISTG.**

Theorem Name	The Great Theorem Prover			AISTG		
	Proven	Time (in sec.)	Length	Proven	Time (in sec.)	Length
S43Wos19	Yes	1	2	Yes	0	10
S44Wos20	Yes	66	21	No	4728	150
S45Wos21	Yes	247	12	No	2525	140
S46Wos22	Yes	29642	16	No	21600	47
S47Wos23	Yes	0	6	No	5251	86*
S48Wos24	Yes	0	6	No	3560	85*
S49Wos25	Yes	0	6	Yes	3	15
S50Wos26	Yes	466	31	No	588	90
S51Wos27	Yes	1	6	No	3835	86
S52Wos28	Yes	14	10	No	4000	258*
S53Wos29	Yes	8	8	Yes	2	15
S54Wos30	Yes	1	11	No	4550	74
S55Wos31	Yes	15	69	No	9390	44
S56Wos32	Yes	0	4	Yes	9	19
S57Wos33	Yes	61	48	Yes	43560	115
Starkey5	Yes	0	4	Yes	0	2
Starkey17	Yes	0	8	Yes	29340	109
Starkey23	Yes	1	7	No	93	65
Starkey26	Yes	0	7	Yes	0	16
Starkey28	Yes	1	7	No	3000	27
Starkey29	Yes	2	7	No	1614	323
Starkey35	Yes	1	7	No	3900	50*
Starkey36	Yes	13	12	No	1800	194
Starkey37	Yes	1	3	No	1112	21
Starkey41	Yes	0	4	Yes	0	4
Starkey55	Yes	0	9	No	7271	308
Starkey65	Yes	0	2	Yes	0	17
Starkey68	Yes	1	8	Yes	0	13
Starkey75	Yes	1	8	No	544	124
Starkey76	Yes	0	3	Yes	0	5
Starkey87	Yes	0	9	Yes	67	55
Starkey100	Yes	0	4	Yes	0	4
Starkey103	Yes	0	9	Yes	160	22
Starkey105	Yes	0	5	Yes	0	8
Starkey106	Yes	0	5	Yes	3	13
Starkey108	Yes	10	28	No	1200	65
Starkey111	Yes	0	5	Yes	0	4
Starkey112	Yes	3	40	No	670	104
Starkey115	Yes	0	7	Yes	68	21
Starkey116	Yes	1	12	Yes	0	14
Starkey118	Yes	3	42	No	9440	175
Starkey121	Yes	1	18	Yes	31317	38

Table 4.1: **The Great Theorem Prover vs AISTG.**

was found or before the program stopped searching (Column 7). The program stopped searching if the number of resolvents on the path to some node in the semantic tree became greater than the size of the clause database (in our case, the size of the database is 5000 clauses).

Table 4.1 shows that The Great Theorem Prover proved all the theorems in the Stickel Test Set, whereas the AISTG proved 47 theorems (i.e. more than 50% of the theorems). Comparing Tables 4.1 and 2.1, it can be seen that better results were obtained using the AISTG on the Stickel Test Set than the canonical semantic tree generator of Chapter 2. Table 4.1 shows that by using the improved ordering of atoms from the Herbrand base, the semantic tree generator solved an additional 15 theorems; it solved all theorems that were solved when using the canonical ordering of atoms from the Herbrand base plus a number of theorems that seemed impossible to solve using the canonical ordering, in particular, **S01burst, S02short, S49wos25, and Starkey65.** This confirms the importance of re-ordering and filtering the Herbrand base in making semantic tree generators practical theorem provers. We strongly believe that semantic tree generators can be driven not only to prove all the Stickel Set theorems, but also to be as strong and efficient as are resolution-dependent theorem provers such as The Great Theorem Prover. In fact, there are theorems which The Great Theorem Prover could not prove in a long time, but our semantic tree generators solved them in a short time. Examples of such theorems will be discussed in the next chapter.

5 Semantic Tree Generation vs Resolution-Refutation

Research in artificial intelligence has pointed out that proving a theorem can be intellectually difficult and that a program that can prove some theorems has "common sense", meaning that it has the ability to make elementary deductions from given facts [Slagle3]. In Chapter 1, Herbrand's fundamental theorem was referred to as a base for many modern proof procedures including those based on the resolution principle of Robinson. Procedures using semantic tree generation for proving theorems are also based on the same theorem. It has been argued in this thesis that the semantic tree generation method can grow to become no less than the other practical methods for detecting unsatisfiable sets of clauses in first-order predicate calculus.

In this chapter, we compare semantic tree generation with resolution-refutation. We identify cases (i.e. theorems) where the former method gave far better results than did the latter one. Conversely, in Section 5.2 we name theorems from the Stickel Test Set for which the proofs obtained by the latter method are more desirable.

One might now wish some insight into the difficulty of the theorems provable with the assistance of semantic tree generators. The following section is

devoted to fulfilling such a wish.

5.1 Generating Semantic Trees as a Proving Method

An obvious question that comes to one's mind with respect to generating semantic trees is "*why should I use this proving method over other existing methods?*". Generating closed semantic trees for theorems in the Stickel Test Set seemed to give interesting results when compared with the proofs obtained by resolution-based procedures such as The Great Theorem Prover. As a matter of fact, the resolution-refutation method is not always superior to the method of generating semantic trees for proving theorems. Certain theorems found in the literature were fabricated by researchers for the purpose of providing a graduated selection of problems for use in testing automated theorem provers [Pelletier1, Spencer1, Urquhart1].

> The difficulty in constructing problems for studying the complexity of the proof system of an ATP[10] is to describe a set of problems whose complexity can independently be characterized in terms of some metric which can be varied and which does not introduce any side effects into the resulting proofs. Various attempts to state such a set have usually focussed on (a) number of clauses, (b) number of symbols, (c) number of distinct symbols.
>
> FJ Pelletier

The following are examples of such problems:

Example 1: Pigeonhole Theorem - cf. [Pelletier1] (page 212)

Problem: Suppose there are n holes and $(n+1)$ objects to put in the holes. Every object is in a hole and no hole contains more than one object.

Let us now state the problem for $n = 3$: *"Each object is in a hole"* becomes:

1. P1 | P2 | P3
2. P4 | P5 | P6

[10] Automated Theorem Prover.

3. P7 | P8 | P9
4. P10 | P11 | P12

"No hole has more than one object in it" becomes

5. ~P1 | ~P4
6. ~P1 | ~P7
7. ~P1 | ~P10
8. ~P4 | ~P7
9. ~P4 | ~P10
10. ~P7 | ~P10
11. ~P2 | ~P5
12. ~P2 | ~P8
13. ~P2 | ~P11
14. ~P5 | ~P8
15. ~P5 | ~P11
16. ~P8 | ~P11
17. ~P3 | ~P6
18. ~P3 | ~P9
19. ~P3 | ~P12
20. ~P6 | ~P9
21. ~P6 | ~P12
22. ~P9 | ~P12

The set of clauses (1) - (22) is inconsistent.

Example 2: Arbitrary Graph Theorems

Problem: Consider a graph (a finite set of vertices, together with a finite set of edges joining pairs of these vertices) with the edges labelled.

Assign a *charge* of 0 or 1 arbitrarily to each vertex in the graph. For each vertex of the graph associate a set of clauses as follows:

1. every label of an edge emanating from that node will occur in each clause of the set of clauses generated from that node.

2. if the node is assigned 0, then the number of negated literals in each of the generated clauses is to be odd. Generate all such clauses for that node.

3. if the node is assigned 1, then the number of negated literals in each of the generated clauses in to be even. Generate all such clauses for that node.

Example 2a: - cf. [Pelletier1] (page 214)

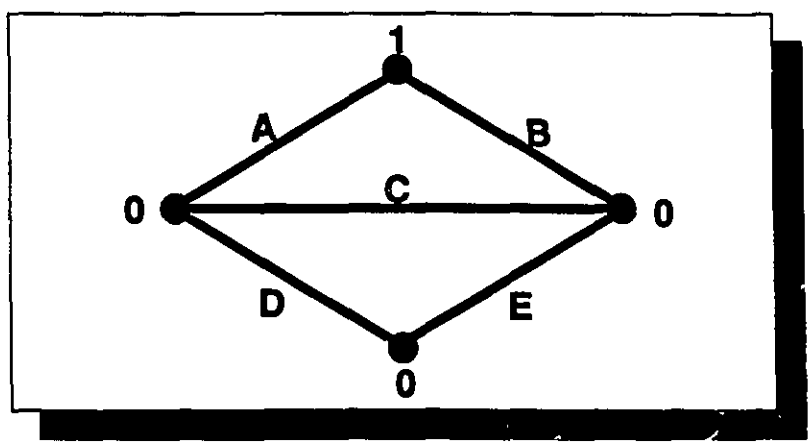

Figure 5.1: **Semi-connected 4-vertices Graph Theorem.**

The set of clauses generated for this example is:

1. A | B
2. ~A | ~B
3. A | C | ~D
4. A | ~C | D
5. ~A | C | D
6. ~A | ~C | ~D
7. B | C | ~E
8. B | ~C | E
9. ~B | C | E
10. ~B | ~C | ~E

11. D | ~E

12. ~D | E

The set of clauses (1) - (12) is inconsistent.

Example 2b: - cf. [Urquhart1] (page 213)

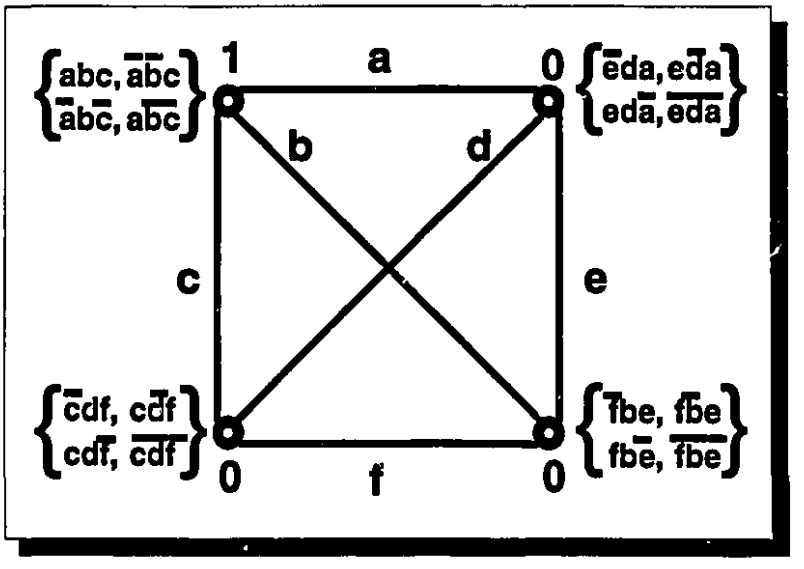

Figure 5.2: **Totally-connected 4-vertices Graph Theorem.**

The set of clauses generated for this example is:

1. a | b | c

2. ~a | ~b | c

3. ~a | b | ~c

4. a | ~b | ~c

5. ~a | d | e

6. a | ~d | e

7. a | d | ~e

8. ~a | ~d | ~e

9. ~c | d | f

10. c | ~d | f

11. c | d | ~f

12. ~c | ~d | ~f
13. ~e | b | f
14. e | ~b | f
15. e | b | ~f
16. ~e | ~b | ~f

The set of clauses (1) - (16) is inconsistent.

Example 3: Foothold Theorems - cf. [Spencer1] (page 580)

Problem: Consider the following set of theorems:

- P | ~P1 | ... | ~Pn
- Pi | ~Ai $\forall$ i = 1, ..., n
- Pi | ~Bi $\forall$ i = 1, ..., n
- Ai | Bi $\forall$ i = 1, ..., n
- ~P

The base clauses of each theorem in this set are inconsistent.

Example 4: Shoe-Boxes Theorems - cf. Unpublished

Problem: The author would like to thank T. Mackling at the Faculty of Engineering in McGill University for providing him with this set of theorems. The symbols A1, ..., An, B1, ..., Bn are constants.

- Equal(x,x)
- ~Equal(x,y) | Equal(y,x)
- ~Equal(x,y) | ~Equal(y,z) | Equal(x,z)
- Equal(x,A1) | ... | Equal(x,An)
- ~Equal(B1,B2)
- ~Equal(B1,B3)
  ⋮
- ~Equal(B1,Bn)
- ~Equal(B2,B3)

Theorem Name	Can. Sem. Tree Gen. Atoms checked	Can. Sem. Tree Gen. Time in sec.	AISTG Atoms checked	AISTG Time in sec.	TGTP Time in sec.
Pigeonhole	12	1	12	0	>100
Arbitrary Graph (a)	5	0	5	0	>100
Arbitrary Graph (b)	6	0	6	0	4
Foothold $n = 1$	4	1	4	0	1
Foothold $n = 2$	7	1	7	0	3
Foothold $n = 3$	10	0	10	0	5
Foothold $n = 4$	13	0	13	1	3
Foothold $n = 5$	16	1	16	2	9
Foothold $n = 6$	19	7	19	12	150
Foothold $n = 7$	22	21	22	51	7860
Foothold $n = 8$	25	175	25	243	28950
Foothold $n = 9$	28	547	28	1062	>144000
Foothold $n = 10$	31	2250	31	4232	>144000
Shoe Boxes $n = 2$	23	3	25	2	137
Shoe Boxes $n = 3$	47	20	46	850	19873
Shoe Boxes $n = 4$	78	349	77	18531	>144000

Table 5.1: **Hard Research Theorems Proved Using Semantic Tree Generators.**

- ~Equal(B2,B4)

  ⋮

- ~Equal(B2,Bn)
- ~Equal(B3,B4)

  ⋮

- ~Equal(Bn-1,Bn)

The base clauses of each theorem in this set are inconsistent.

We tried some of these theorems on our semantic tree generators. It turned out that the semantic tree generators gave a far greater performance than did The Great Theorem Prover. Table 5.1 displays the result of proving such theorems. For most theorems of this type, The Great Theorem Prover kept searching for the proofs for a long time (in some cases for more than a day) that we decided to discontinue the search. The reason for this is either too many useless resolvents generated or the generated resolvents have to many literals in them. The table displays the result of proving these theorems using the canonical semantic tree generator and the AISTG. Column 1 shows the name of the theorem. Column 2 shows the number of atoms canonically enumerated

from the Herbrand base that are sufficient to prove the theorem. Column 3 shows the execution time in seconds that it took the canonical semantic tree generator to prove the theorem. Column 4 shows the number of atoms selectively chosen from the Herbrand base by the AISTG using the fewest-literals strategy for proving the theorem. Column 5 shows the execution time in seconds that it took the AISTG to prove the theorem. Column 6 shows the execution time in seconds that it took TGTP to prove the theorem. For this experiment, we used the same computer that we had used for the previous experiments.

Although the above-illustrated examples present considerable difficulties for resolution-based theorem provers, this fact does not reflect any real inherent difficulty in the problems, but rather the inefficient way in which the resolution procedure deals with theorems of this kind. This fact is demonstrated by the existence of short refutations for these examples in axiomatic systems for propositional calculus such as semantic tree generators.

5.2 When to Avoid Generating Semantic Trees for Proving Unsatisfiability

Researchers who have tried to compile lists of problems for automated theorem provers in the past have discovered that the production of such lists is difficult. One of the reasons for this difficulty is that what seems to be "easy" for one system might not be for another. Indeed, this is exactly the case between semantic tree generators and resolution-refutation theorem provers.

In the previous section, several theorems were presented for which the proofs obtained by semantic tree generators were vastly more efficient than were those obtained by resolution-refutation theorem provers. To complete the picture, in this section we discuss the converse situation. We identify theorems which are considered trivial for resolution-refutation theorem provers, but they seem to be impossible to solve using semantic tree generators. Table 2.1 appeared earlier in Chapter 2 includes theorems of this type. Examples of such

theorems are: **S01burst, S05haspart2, S18DBABHP, S19APABHP, S39Wos15, S46Wos22, S50Wos26, Starkey28, and Starkey75**. Yet, in order to express this fact more clearly, let us consider the following set of base clauses:

Axioms:

1. P(f(g(a,b,c),d)) | Q(f(a,b))
2. ~P(x)

Negated Theorem:

3. ~Q(f(x,y))

A resolution-refutation proof for this theorem was obtained by The Great Theorem Prover in less than a second. However, proving this theorem by using the canonical semantic tree generator appeared to be much harder. In fact, the canonical semantic tree generator could not prove this theorem for the following obvious reason:

The canonical enumeration of elements from the Herbrand universe is:
HU = { a, b, c, d, f(a,a), f(a,b), f(a,c), f(a,d), f(b,a), f(b,b), f(b,c), f(b,d), f(c,a), f(c,b), f(c,c), f(c,d), f(d,a), f(d,b), f(d,c), f(d,d), g(a,a,a), ... }.

The canonical enumeration of atoms from the Herbrand base is:
HB = { P(a), Q(a), P(b), Q(b), P(c), Q(c), P(d), Q(d), P(f(a,a)), Q(f(a,a)), P(f(a,b)), Q(f(a,b)), P(f(a,c)), Q(f(a,c)), P(f(a,d)), Q(f(a,d)), P(f(b,a)), Q(f(b,a)), P(f(b,b)), Q(f(b,b)), P(f(b,c)), Q(f(b,c)), P(f(b,d)), Q(f(b,d)), P(f(c,a)), Q(f(c,a)), P(f(c,b)), Q(f(c,b)), P(f(c,c)), Q(f(c,c)), P(f(c,d)), Q(f(c,d)), P(f(d,a)), Q(f(d,a)), P(f(d,b)), Q(f(d,b)), P(f(d,c)), Q(f(d,c)), P(f(d,d)), Q(f(d,d)), P(g(a,a,a)), Q(g(a,a,a)), ... }.

$| HU_0 | = 4$, $| HU_1 | = 84$, $| HU_2 | = 599764$, ... etc.
$| HB_0 | = 8$, $| HB_1 | = 168$, $| HB_2 | = 1.19 \times 10^6$, ... etc.

In order to prove this theorem using a canonical semantic tree generator, at least 168 atoms must be checked before a closed semantic tree is generated. To be exact, the term f(g(a,b,c),d) appears after 2255 terms in the canonical

enumeration of the Herbrand universe. Consequently, 4511 Herbrand base atoms must be checked before a closed semantic tree can be generated; this is beyond the capability of the canonical semantic tree generator. However, these figures are not meant to discredit the semantic tree generation as a theorem-proving method. For example, filtering the Herbrand base of the above theorem, as proposed in Method I for improving the practicality of generating semantic trees, leaves only two elements in the Herbrand base to be checked by the generator, namely, P(f(g(a,b,c),d) and Q(f(a,b)). This theorem can be used as another example for encouraging the use of improved semantic tree generators as practical theorem provers.

In summary, our experimental study confirms that the semantic trees method can be as good as (in some cases even better than) other methods for proving unsatisfiability of sets of clauses, including the resolution-refutation method. This chapter has shown that this method can sometimes be a better choice for solving certain hard theorems than what is considered the best and most powerful of all theorem-proving methods, which is resolution-refutation.

6 Conclusion

This dissertation has investigated the use of semantic trees in automated theorem proving. In it, we studied the effectiveness of Herbrand's procedure on theorems such as those in the Stickel Test Set. Additionally, we looked at more effective ways of ordering the atoms of the Herbrand base that are used for generating semantic trees, and we showed that a larger set of theorems could be proved. In what follows, we summarize the findings of the previous chapters and suggest various techniques for further advancements in generating semantic trees for proofs of unsatisfiability.

6.1 Concluding Remarks

Focusing on semantic trees and on their role in automated theorem proving, this thesis has demonstrated both the equivalence of semantic trees and resolution-refutation proof trees and thus the equivalence of semantic tree generators and resolution-refutation theorem provers. A system for using semantic trees in proving unsatisfiability was illustrated in Chapter 2. The system included generating a closed semantic tree from a given resolution-

refutation proof of an unsatisfiable set of clauses, as well as the extraction of a resolution-refutation proof from a closed semantic tree of that set. Chapter 2 also described canonical semantic trees and provided examples of constructing canonical semantic trees of unsatisfiable clauses. We closed Chapter 2 with an experiment measuring the performance of a canonical semantic tree generator in proving theorems from the Stickel Test Set.

Theorem-proving on the computer, using procedures based on semantic trees was examined in Chapter 3 with a view towards improving the efficiency and widening the range of practical applicability of automated theorem proving. Three methods for improving the practicality of generating semantic trees for proofs of unsatisfiability were considered: filtering the Herbrand base; proposing control strategies for selectively choosing atoms from the Herbrand base; and interactively manipulating the order by which the atoms appear in the enumeration of the Herbrand base. These methods we implemented in a semantic tree generator and tested on theorems from the Stickel Test Set.

Chapter 4 presented AISTG: An Improved Semantic Tree Generator embodying the three methods suggested in Chapter 3 for improving the practicality of generating semantic trees for proofs of unsatisfiability. The chapter described modules and layouts of the AISTG program as well as the flow of control in the program. In addition, it outlined principal features of the program and some of its limitations. We closed Chapter 4 with an experiment comparing the performance of the AISTG with The Great Theorem Prover on the Stickel Test Set.

Chapter 5 addressed the consideration of generating semantic trees as an alternative method for proving unsatisfiability of sets of clauses. On the one hand, the chapter presented classes of theorems for which the semantic tree generators are expected to perform at least as good as, if not better than, resolution-refutation theorem provers. On the other hand, it provided examples of theorems for which the semantic tree proofs are less desirable than the resolution-refutation proofs.

6.2 Open Problems

The following are suggestions for further improvements in generating semantic trees for proofs of unsatisfiability:

- In Chapter 3 we have introduced control strategies for selectively choosing atoms from the Herbrand base of a given set of clauses. Any additional study on the effect of control strategies on the practicality of generating semantic trees would surely contribute to the improvement of generating semantic trees for proofs of unsatisfiability.

- Despite the importance of search strategies, most research in automatic theorem proving has concentrated on developing new inference systems which are either more powerful or more restrictive than those already existing [Kowalski3]. Other control strategies and heuristics for eliminating or re-ordering atoms in the Herbrand base should be considered [Bledsoe1, Kowalski1, Kowalski3, Norton1, Siklossy1, Slagle1].

- Selectively choosing atoms from the Herbrand base has improved the practicality of generating semantic trees for proofs of unsatisfiability. Since domain knowledge has shown its importance in many AI applications, having a user interactively select atoms from the Herbrand base for building semantic trees should be further explored.

- Two decades ago, efficient general-purpose theorem-proving systems based on resolution without equality were developed and used for proving theorems in first-order predicate logic. Unfortunately, they were not able to prove anything very complicated, and additions were sought which would make them more powerful [Chang1]. One way to increase their power was to incorporate equality into these logical systems. For instance, the explicit use of equality axioms, the application of paramodulation and E-resolution, and resolution by unification and equality [Plotkin1, Robinson2, Slagle1]. The AISTG has no identity in its system other than

as an ordinary predicate (i.e. the equality axioms). The introduction of equality into the AISTG would undoubtedly increase its efficiency by allowing it to prove more complicated theorems. How? When the axioms of reflexivity, symmetry, and transitivity for the equality are "built-in", any clause containing a substitution instance of (i.e. subsumed by) one of these clauses is a tautology which can be ignored [Caines1].

- There are further search methods of the same general sort as semantic tree generation, which are less simple than those discussed in this dissertation. An extension of our research is planned in which the theoretical framework developed here will be used as a basis for more extensive treatments of search methods based on semantic trees and of the design of semantic tree generators.

- A parallel semantic tree generator should be explored. There seem to be many design alternatives when parallelizing the generation of these trees. The second proposed method for improving the performance of semantic tree generators is particularly appropriate for implementation on multiprocessor computers.

- Lastly, a theorem prover that combines a semantic tree generator with a resolution-refutation proof searcher seems to offer interesting possibilities.

It is our belief that semantic trees have been unjustly overlooked by researchers in proving unsatisfiability of clauses. With this presentation, we hope that we have helped to magnify the underestimated role of semantic trees in automated theorem proving, and thereby have undertaken a first step toward exploiting the real potential of semantic tree generators as more practical than theoretical tools for proving first-order logic theorems.

Bibliography

[Almulla1] M. Almulla and M. Newborn, The practicality of generating semantic trees for proofs of unsatisfiability, *submitted to The Tenth Biennial Conference on AI and Cognitive Science, Hybrid Problems, Hybrid Solutions*, Sheffield, England, April 1995.

[Almulla2] M. Almulla, M. Newborn and B. Patrick, An upper bound on the time complexity of iterative-deepening-A*, *Annals of Mathematics and Artificial Intelligence*, Switzerland, V. 5, N. 1-2, pp. 265-78, May 1992.

[Anderson1] R. Anderson and W. Bledsoe, A linear format for resolution with merging and a new technique for establishing completeness, *Journal of the Association for Computing Machinery*, V. 17, N. 3, pp. 525-534, July 1970.

[Andrews1] P. B. Andrews, Theorem proving via general matings, *Journal of the Association for Computing Machinery*, V. 28, pp. 193-214, 1981.

[Bagai1] R. Bagai, V. Shanbhogue, J. M. Zytkow and S. C. Chou, Automatic theorem generation in plane geometry, *Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93*, Norway, Proceedings, pp. 415-424, June 1993.

[Bledsoe1] W. W. Bledsoe, Splitting and reduction heuristics in automatic theorem proving, *Artificial Intelligence*, V. 2, pp. 57-78, 1971.

[Boyer1] R. S. Boyer, *Locking: A Restriction of Resolution*, Ph.D. Thesis, University of Texas at Austin, Texas, 1971.

[Boyer2] R. S. Boyer and J. S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.

[Broy1] M. Broy, On the Herbrand-kleene universe for nondeterministic computations, *Theoretical Computer Science*, Netherlands, V. 36, N. 1, pp. 1-19, March 1985.

[Caines1] P. E. Caines, T. Mackling and Y. J. Wei, Logical control via automatic theorem proving: COCOLOG fragments implemented in Blitzensturm 5.0, *Proceedings of the American Control Conference*, San Francisco, pp. 1209-13, 1993.

[Chang1] C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[Chang2] C. L. Chang, Theorem proving by generation of pseudo-semantic trees, *Div. of Comput. Res. and Technol., Nat. Inst. of Health, Bethesda*, Maryland, 1971.

[Chang3] C.L. Chang, The unit proof and the input proof in theorem proving, *Journal of the Association for Computing Machinery*, V. 17, pp. 698-707, 1970.

[Chu1] H. Chu and D. A. Plaisted, Model finding strategies in semantically guided instance-based theorem proving, *Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93*, Norway, Proceedings, pp. 19-28, June 1993.

[Emden1] M. H. V. Emden and R. A. Kowalski, The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, V. 23, N. 4, pp. 733-42, October 1976.

[Fermuller1] C. Fermuller, A. Leitsch, T. Tammet and N. Zamov, *Resolution Methods for the Decision Problem*, Lecture Notes in Artificial Intelligence, V. 679, Berlin, Germany, Springer-Verlag, 1993.

[Fleisig1] S. Fleisig, D. Loveland, A. Smiley III and D. Yarmush, An implementation of the model elimination proof procedure, *Journal of the Association for Computing Machinery*, V. 28, N. 12, pp. 124-139, 1974.
n

[Gelernter1] H. Gelernter, Realization of a geometry theorem proving machine, *Proc. IFIP Congress*, pp. 273-282, 1959.

[Gilmore1] P. C. Gilmore, A proof method for quantification theory; its justification and realization, *IBM J. Res. Develop.*, pp. 28-35, 1960.

[Hayes1] J. P. Hayes, *Semantic Trees: New Foundations for Automatic Theorem Proving*, Ph.D. Dissertation, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, 1973.

[Herbrand1] J. Herbrand, *On the Consistency of Arithmetic, "From Frege to Godel: a Source Book in Mathematical Logic"*, Edited by Jean Van Heijenoort, Harvard University Press, Cambridge, Massachusetts, 1931.

[Herbrand2] J. Herbrand, *Logical Writings, A Translation of the "Ecrits Logiques"*, Edited by Jean Van Heijenoort, Harvard University Press, Cambridge, Massachusetts, 1930.

[Hsiang1] J. Hsiang and M. Rusinowitch, Proving refutational completeness of theorem-proving strategies: the transfinite semantic tree method, *Journal of the Association for Computing Machinery*, V. 38, N. 3, pp. 559-587, July 1991.

[Karl1] M. Karl, The Markgraf Karl Refutation Procedure. Memo SEKI-MK-84-01, Fachbereich Informatik, Univeritat Kaiserslautern, Kaiserslautern, West Germany, January 1984.

[Klahr1] Ph. Klahr and D. Waterman, *Expert Systems Techniques, Tools and Applications*, Addison-Wesley Publishing Company, 1986.

[Kowalski1] R. A. Kowalski, *Logic for Problem Solving*, Elsevier North Holland, New York, 1979.

[Kowalski2] R. A. Kowalski, Search strategies for theorem proving, *Machine Intelligence 5*, pp. 87-101, 1970.

[Kowalski3] R. A. Kowalski, Linear resolution with selection function, *Metamathematics unit*, Edinburgh University, Scotland, 1970.

[Kowalski4] R. A. Kowalski and J. P. Hayes, Semantic trees in automatic theorem proving, *Machine Intelligence 4*, pp. 87-101, 1969.

[Letz1] R. Letz, S. Bayerl and W. Bibel, SETHEO, a high performance theorem prover, *Journal of Automated Reasoning*, V. 8, pp. 183-213, 1992.

[LiMin1] F. LiMin, *Neural Networks in Computer Intelligence*, McGraw-Hill Series in Computer Science, McGraw-Hill Inc., 1994.

[Loveland1] D. W. Loveland, Theorem provers combining model elimination and resolution, *Machine Intelligence 4*, pp. 73-86, 1984.

[Loveland2] D. W. Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland Publishing Company, 1978.

[Loveland3] D. W. Loveland, A unifying view of some linear Herbrand procedures, *Journal of the Association for Computing Machinery*, V. 19, N. 2, pp. 366-384, April 1972.

[Loveland4] D. W. Loveland, A linear format for resolution, *Proc. IRIA Symp. Automatic Demonstration*, Versailles, France, Springer-Verlag, pp. 147-162, 1970.

[Loveland5] D. W. Loveland, Some linear Herbrand proof procedures: an analysis, Department of Computer Science, Carnegie-Mellon University, 1970.

[Loveland6] D. W. Loveland, A simplified format for the model elimination theorem-proving procedure, *Journal of the Association for Computing Machinery*, V. 16, N. 3, pp. 349-363, 1969.

[Loveland7] D. W. Loveland, Mechanical theorem proving by model elimination, *Journal of the Association for Computing Machinery*, V. 15, pp. 236-251, 1968.

[Luger1] G. Luger and W. Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Second edition, The Benjamin/Cummings Publishing Company, 1993.

[Luckham1] D. Luckham, Refinements in resolution theory, *Proc. IRIA Symp. Automatic Demonstration*, Versailles, France, Springer-Verlag, pp. 163-190, 1970.

[Luckham2] D. Luckham, Some tree-pairing strategies for theorem proving, *Machine Intelligence 3*, pp. 95-112, 1968.

[Manna1] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Series in Computer Science, McGraw-Hill Inc., 1974.

[McCune1] W. McCune, Otter 2.0 Users Guide, ANL-90/9, Argonne National Laboratory, Mathematics and Computer Science Division, 1990.

[McCharen1] J. D. McCharen, R. A. Overbeek and L. A. Wos, Problems and experiments for and with automated theorem-proving programs, *IEEE Transactions on Computers*, V. C-25, N. 8, pp. 773-782, 1976.

[Michie1] D. Michie, R. Ross and G. Shannan, G-deduction. *Machine Intelligence*, V. 7, New York, pp. 141-165, 1972.

[Newborn1] M. Newborn, *The Great Theorem Prover Version 2*, Newborn Software, 1994.

[Newborn2] M. Newborn, Y. Qingxun and H. Zhang, Test Results for the Great Theorem Prover, Technical Report - SOCS91.9, September 1991.

[Nilsson1] N. J. Nilsson and M. R. Geneserth, *Logical Foundation of Artificial Intelligence*, Morgan Kauffman Publishers, Inc., 1987.

[Nilsson2] N. J. Nilsson, *Principles of Artificial Intelligence*, Morgan Kauffman Publishers, Inc., 1980.

[Nilsson3] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill Series in Computer Science, McGraw-Hill Inc., 1971.

[Norton1] L. M. Norton, Experiments with a heuristic theorem proving for the predicate calculus with equality, *Artificial Intelligent* V. 2, pp. 261-284, 1971.

[Norton2] L. M. Norton, *Adept-A Heuristic Program for Proving Theorems of Group Theory*, Ph.D. Thesis, M.I.T., Cambridge, Mass., 1966.

[Nossum1] R. Nossum, Automated theorem proving methods, *Nordisk Tidskrift for Informationsbehandling (BIT)*, V. 25, N. 1, pp. 51-64, 1985.

[Nossum2] R. Nossum, *Decision Algorithms for Program Verification*, University of Oslo, 1984.

[Passos1] E. P. Passos, R. L. De Carvalho and M. M. Pion, Interactive system to construct minimal model on the Herbrand universe, *Proceedings of a Symposium Organized by the Austrian Society of Cybernetic Studies*, pp. 337-342, 1982.

[Passos2] E. P. Passos, R. L. De Carvalho and S. R. Peixoto, Communication predicates: a complete strategy for resolution-based theorems provers an Evaluation of an Implementation, *Proceedings of the Fourth International Congress of Cybernetics & Systems*, Netherlands, pp. 60-62, August 1978.

[Pelletier1] F. J. Pelletier, Seventy-five problems for testing automatic theorem provers, *Journal of Automated Reasoning*, V. 2, pp. 191-216, 1986.

[Peterson1] G. Peterson, A technique for establishing completeness results in theorem proving with equality, *SIAM Journal on Computing*, V. 12, N. 1, pp. 82-100, February 1983.

[Plotkin1] G.D. Plotkin, Building-in equational theories, *Machine Intelligence*, V. 7, pp. 73-89, 1972.

[Putnam1] H. Putnam and M. Davis, A computing procedure for quantification theory, *Journal of the Association for Computing Machinery*, V. 7, pp. 201-215, 1960.

[Qian1] L. R. Qian, Semantic mappings on Herbrand base, *Chinese Academy of Sciences (Science Bulletin)*. English Edition, V. 27, N. 10, pp. 1042-1045, 1982.

[Quaife1] A. Quaife, Automated development of Tarski's geometry, *Journal of Automated Reasoning*, V. 5, pp. 97-118, 1989.

[Reboh1] R. Reboh, B. Raphael, R. Yates, R. Kling and C. Verlarde, Study of automatic theorem-proving programs, Technical Note 75, Artificial Intelligence Center, Stanford Research Institute, Ca. November 1972.

[Reiter1] R. Reiter, Two results on ordering for resolution with merging and linear format, *Journal of the Association for Computing Machinery*, V. 18, pp. 630-646, October, 1971.

[Robinson1] J. A. Robinson, Computational logic: the unification computation, *Machine Intelligence 6*, pp. 63-72, 1971.

[Robinson2] J. A. Robinson and L. Wos, Paramodulation and theorem proving in first order theories with equality, *Machine Intelligence*, V. 4, pp. 135-150, 1969.

[Robinson3] J. A. Robinson, The generalized resolution principle, *Machine Intelligence 3*, pp. 77-94, 1968.

[Robinson4] J. A. Robinson, A review of automatic theorem proving, *Proc. Symp. Appl. Math. Amer. Math. Soc. 19*, pp. 1-18, 1967.

[Robinson5] J. A. Robinson, A machine-oriented logic based on the resolution principle, *Journal of the Association for Computing Machinery*, V. 12, N. 1, pp. 23-41, 1965.

[Robinson6] J. A. Robinson, Automatic deduction with hyper-resolution, *International Journal of Computing Machinery*, V. 1, pp. 227-234, 1965.

[Robinson7] J. A. Robinson, Theorem-proving on the computer, *Journal of the Association for Computing Machinery*, V. 10, pp. 163-174, April 1963.

[Siklossy1] L. Sikloosy and V. Marinov, Heuristic search and exhaustive search, *Proc. 2nd International Conference on Artificial Intelligence*, London, pp. 601-607, 1971.

[Slagle1] J. R. Slagle, Automatic theorem proving with built-in theories including equality, partial ordering and sets, *Journal of the Association for Computing Machinery*, V. 19, N. 1, pp. 120-135, January 1972.

[Slagle2] J. R. Slagle and C. D. Farrell, Experiments in automatic learning for a multipurpose heuristic program, *Comm. Association for Computing Machinery*, V. 14, pp. 91-99, 1971.

[Slagle3] J. R. Slagle, Automatic theorem proving with renamable and semantic resolution, *Journal of the Association for Computing Machinery*, V. 14, N. 4, pp. 687-697, October 1967.

[Spencer1] B. E. Spencer, Avoiding duplicate proofs, *Logic Programming. Proceedings of the 1990 North American Conference*, Austin, TX, pp. 569-84, October 1990.

[Starkey1] J. D. Starkey and J. D. Lawrence. Experimental tests of resolution based theorem-proving strategies. Technical Report, Computer Science Department, Washington State University, Washington, April 1974.

[Stickel1] M. E. Stickel, Automated theorem proving research in the fifth generation computer systems project: model generation theorem provers, *Future Generation Computer Systems*, V. 9, N. 2, pp. 143-52, July 1993.

[Stickel2] M. E. Stickel, A Prolog technology theorem prover: implementation by an extended Prolog compiler, *Journal of Automated Reasoning*, V. 4, pp. 353-380, 1988.

[Stickel3] M. E. Stickel, An analysis of consecutively bounded depth-first search with applications in automated deduction, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, Cal., V. 2, pp. 1073-1075, August 1985.

[Stoll1] R. Stoll. *Set Theory And Logic*, W. H. Freeman and Company, A series of books in mathematics, 1963.

[Tanimoto1] S. L. Tanimoto, *The Elements of Artificial Intelligence: An Introduction Using LISP*, Computer Science Press, Rockville, Maryland, 1987.

[Wang1] S. Wang and P. E. Caines, Automated reasoning with function evaluation for COCOLOG with examples, *The 31st IEEE Conference on Decision and Control*, Tusca, AZ, December 1992. Complete version: Research Report N. 1713, INRIA-Sophia Antipolis, 1992.

[Wilson1] G. Wilson and J. Minker, Resolution, refinements, and search strategies: a comparative study, *IEEE Transactions on Computers* C-25, N. 8, pp. 782-801, August 1976.

[Wos1] L. Wos, Automated reasoning answers open questions, *Notices of the American Math. Society*, pp. 15-26, January 1993.

[Wos2] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications*, 2nd Edition, McGraw-Hill Series in Computer Science, McGraw-Hill Inc., New York, 1992.

[Wos3] L. Wos, *Automated Reasoning: 33 Basic Research Problems*, Englewood Cliffs, New Jersey, Prentice-Hall, 1988.

[Wos4] L. Wos, Efficiency and completeness of the p set-of-support strategy in theorem proving, *Journal of the Association for Computing Machinery*, V. 12, pp. 536-541, 1965.

[Wos5] L. Wos, D. Carson and J. A. Robinson, The unit preference strategy in theorem proving. *Proc. AFIPS 1964 Fall Joint Computer Conference*, V. 26, pp. 616-621, 1964.

[Urquhart1] A. Urquhart, Hard examples for resolution, *Journal of the Association for Computing Machinery*, V. 34, N. 1, pp. 209-219, 1987.

Appendix A: Proving the Stickel Test Set using the AISTG

(PP. 97)

	Strategy 1: Herb. Proc.			Strategy 2: FLS			Strategy 3: SOS			Strategy 4: LF			Strategy 5: FLS+SOS			Strategy 6: FLS+LF			Best
Theorem	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	
burst	No	66	65m	Yes	45	9s	No	70	65m	Yes	17	1s	No	96	65m	Yes	45	22s	4
short	No	90	500s	Yes	11	0	No	105	500s	No	87	31s	No	43	120s	Yes	14	1s	2
prime	Yes	187	5.5h	Yes	31	217s	Yes	28	154s	Yes	187	5.5h	Yes	21	9s	Yes	31	290s	5
haspa1	No	252	54s	No	91	20s	No	281	55s	No	33	15m	No	270	55s	No	64	13s	0
haspa2	No	93	93s	No	96	15m	No	120	95s	No	64	5s	No	123	95s	No	65	17s	0
ances	Yes	6	1s	Yes	6	1s	Yes	6	1s	Yes	6	1s	Yes	6	1s	Yes	6	1s	Any
num1	Yes	177	4h	Yes	17	2s	Yes	218	4h	Yes	22	96s	Yes	209	4h	Yes	17	2s	2,6
group1	No	96	40m	No	96	446s	No	56	83s	No	47	40s	No	56	83s	No	56	83s	0
group2	Yes	59	150m	Yes	26	40s	Yes	109	150m	Yes	58	344s	Yes	96	163m	Yes	16	4s	6
ew1	Yes	5	0	Yes	5	0	Yes	5	0	Yes	5	0	Yes	5	0	Yes	5	0	Any
ew2	Yes	3	1s	Yes	3	1s	Yes	3	1s	Yes	3	1s	Yes	3	1s	Yes	3	1s	Any
ew3	Yes	5	1s	Yes	5	1s	Yes	5	1s	Yes	5	1s	Yes	5	1s	Yes	5	1s	Any
rob1	Yes	13	10s	Yes	10	0	Yes	9	0	Yes	8	0	Yes	9	0	Yes	10	0	4
rob2	Yes	54	156m	Yes	25	32s	Yes	108	4.3h	Yes	82	156m	Yes	90	186m	Yes	16	2s	6
michie	Yes	182	5.4h	Yes	282	5.4h	Yes	187	5.4h	Yes	196	5.4h	Yes	187	5.4h	Yes	187	5.4h	1
qw	Yes	3	1s	Yes	3	0	Yes	3	1s	Yes	3	0	Yes	3	0	Yes	3	1s	Any
mqw	Yes	3	1s	Yes	3	0	Yes	3	0	Yes	3	0	Yes	3	0	Yes	3	1s	Any
DBABH	No	62	>4h	No	114	94s	No	102	4h	No	83	17s	No	102	4h	No	98	66s	0
APABH	No	271	60s	No	95	190s	No	73	35s	No	300	63s	No	93	44s	No	94	435s	0
fleisig 1	No	100	156s	Yes	30	94s	Yes	26	82s	Yes	36	74s	Yes	26	16s	Yes	30	19s	3,5
fleisig 2	No	150	248s	Yes	34	32s	Yes	42	464s	Yes	26	8s	Yes	42	408s	Yes	34	30s	4
fleisig 3	No	70	62m	No	91	104s	No	64	747s	No	112	144s	No	95	122s	No	83	90s	0
fleisig 4	No	42	51m	No	42	270s	No	42	51m	No	76	115s	No	42	51m	No	22	453s	0
fleisig 5	No	42	57m	No	25	253s	No	42	57m	No	76	158s	No	42	57m	No	22	457s	0

h = hours; m = minute; s = second.

(PP. 98)	Strategy 1: Herb. Proc.			Strategy 2: FLS			Strategy 3: SOS			Strategy 4: LF			Strategy 5: FLS+SOS			Strategy 6: FLS+LF			Best
Theorem	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	
wos 1	No	95	21m	No	64	50s	No	70	52s	No	99	156s	No	70	70s	No	77	67s	0
wos 2	No	70	120m	No	60	50s	No	63	30s	No	70	91s	No	81	75s	No	47	346s	0
wos 3	Yes	8	0	Yes	13	1s	Yes	62	14s	Yes	14	4s	Yes	93	64s	Yes	19	4s	1
wos 4	Yes	72	571s	Yes	160	25m	Yes	38	78s	Yes	119	675s	Yes	119	809m	Yes	121	733s	3
wos 5	No	101	1.25h	No	74	179s	No	63	40s	No	70	56s	No	126	207s	No	126	174s	0
wos 6	No	56	2.10h	No	45	571s	No	57	2h	No	64	2h	No	85	2h	No	92	2h	0
wos 7	Yes	74	30m	Yes	46	118s	Yes	136	35m	Yes	116	32m	Yes	112	30m	Yes	111	30m	2
wos 8	No	94	15m	No	78	261s	No	80	41s	No	50	79s	No	54	33s	No	26	7s	0
wos 9	No	130	2h	No	158	2h	No	40	93s	No	130	2h	No	56	31s	No	150	2h	0
wos 10	Yes	70	119m	Yes	34	266s	Yes	131	134m	Yes	78	119m	Yes	117	119m	Yes	108	119m	2
wos 11	No	105	3.5h	No	140	3.5h	No	149	3.5h	No	112	3.5h	No	116	3.5h	No	133	3.5h	0
wos 12	Yes	24	12s	Yes	11	0	Yes	10	0	Yes	11	1s	Yes	8	0	Yes	11	1s	5
wos 13	No	103	60m	Yes	16	1s	No	82	42m	No	51	69s	Yes	10	0	Yes	22	6s	5
wos 14	Yes	30	93s	Yes	23	3s	Yes	17	2s	Yes	80	186s	Yes	16	3s	Yes	93	313s	5
wos 15	No	108	60m	No	92	188s	No	55	20s	No	108	60m	No	52	94s	No	46	40s	0
wos 16	Yes	24	36s	Yes	100	397s	Yes	106	108s	Yes	22	20s	Yes	75	71s	Yes	21	11s	6
wos 17	Yes	69	5h	Yes	135	5h	Yes	93	5h	Yes	107	5h	Yes	106	5h	Yes	105	5h	1
wos 18	No	108	2.10h	Yes	8	0	No	65	89s	No	116	2.1h	No	75	84s	Yes	8	0	2,6
wos 19	No	83	42m	Yes	10	0	No	96	43m	No	90	42m	No	72	234s	Yes	13	1s	2
wos 20	No	102	51m	No	94	558s	No	117	51m	No	109	51m	No	85	261s	No	59	148s	0
wos 21	No	79	58m	No	95	539s	No	91	55s	No	31	25s	No	53	198s	No	26	12s	0
wos 22	No	35	50s	No	54	286s	No	62	54s	No	54	166s	No	53	22s	No	41	32s	0
wos 23	No	43	5m	No	53	527s	No	34	15m	No	5	0	No	50	192s	No	73	5m	0
wos 24	No	43	5m	No	53	249s	No	34	15m	No	5	0	No	25	668s	No	20	15s	0
wos 25	No	43	5m	Yes	15	3s	No	83	113s	No	3	0	No	116	42s	Yes	13	3s	6

h = hours; m = minute; s = second.

(PP. 99) Strategy 1: Herb. Proc. Strategy 2: FLS Strategy 3: SOS Strategy 4: LF Strategy 5: FLS+SOS Strategy 6: FLS+LF Best

Theorem	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	
wos 26	No	72	36m	No	86	340s	No	85	319s	No	106	54s	No	69	105s	No	82	109s	0
wos 27	No	48	63m	No	40	20s	No	33	26s	No	55	63m	No	57	223s	No	33	19s	0
wos 28	No	41	262s	No	20	20s	No	46	9s	No	47	262s	No	44	28s	No	46	12s	0
wos 29	No	73	591s	Yes	15	2s	No	55	46s	No	76	591s	No	28	18s	No	86	62	2
wos 30	No	90	4h	No	41	83s	No	34	29s	No	6	1s	No	66	162s	No	64	189s	0
wos 31	No	69	2.1h	No	60	290s	No	38	15m	No	20	15m	No	42	871s	No	37	15m	0
wos 32	Yes	31	756s	Yes	19	9s	Yes	38	200s	Yes	88	903s	Yes	68	23m	Yes	17	16s	6
wos 33	Yes	70	12h	Yes	115	12.1h	Yes	125	12.1h	Yes	120	12h	Yes	128	14h	Yes	107	12.1h	1
strk 5	Yes	2	0	Yes	2	0	Yes	2	0	Yes	2	0	Yes	2	0	Yes	2	0	Any
strk 17	Yes	69	8h	Yes	109	8.15h	Yes	18	1s	Yes	27	99s	Yes	17	2s	Yes	134	8.1h	5
strk 23	No	85	55m	No	65	93s	No	0	0	No	65	207s	No	64	317s	No	66	349s	0
strk 26	Yes	21	0	Yes	16	0	Yes	10	0	Yes	78	182s	Yes	70	128s	Yes	70	128s	3
strk 28	No	225	22m	No	27	15m	No	90	42s	No	11	1s	No	49	83s	No	83	50m	0
strk 29	No	250	33m	No	60	26s	No	89	36s	No	11	1s	No	59	42s	No	59	44s	0
strk 35	No	30	202s	No	27	257s	Nc	48	158s	No	32	405s	No	29	342s	No	25	18s	0
strk 36	No	131	29m	No	74	112s	No	59	382s	No	135	29m	No	93	52s	No	94	76s	0
strk 37	No	52	465s	No	30	111s	No	22	2s	No	55	192s	No	29	25s	No	29	25s	0
strk 41	Yes	8	0	Yes	4	0	Yes	6	0	Yes	57	281	Yes	28	6s	Yes	28	6s	2
strk 55	No	245	2h	No	64	19s	No	96	64s	No	269	2h	No	26	6s	No	39	10s	0
strk 65	No	250	25m	Yes	17	0	No	82	19s	No	45	16s	No	114	43s	Yes	16	1s	6
strk 68	No	257	20m	Yes	13	0	Yes	3	0	No	106	88s	Yes	3	0	No	85	115s	3,5
strk 75	No	258	668s	No	94	61s	No	59	15s	No	120	186s	No	89	125s	No	85	112s	0
strk 76	Yes	17	20s	Yes	5	0	Yes	10	1s	No	76	52s	Yes	16	21s	Yes	66	114s	2
strk 87	No	250	32m	Yes	55	67s	No	22	3s	No	45	12s	No	102	57s	Yes	58	577s	2
strk 100	Yes	24	741s	Yes	4	0	Yes	16	7s	Yes	8	0	Yes	18	40s	Yes	6	0	2

h = hours m = minute; s = second.

(PP. 100)

Theorem	Strategy 1: Herb. Proc.			Strategy 2: FLS			Strategy 3: SOS			Strategy 4: LF			Strategy 5: FLS+SOS			Strategy 6: FLS+LF			Best
	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	Proven	# Elements	Time	
strk 103	Yes	37	7.3h	Yes	22	160s	Yes	12	1s	Yes	109	7.3h	Yes	9	1s	Yes	25	313s	5
strk 105	Yes	66	12h	Yes	8	0	Yes	19	16s	Yes	22	25m	Yes	16	6s	Yes	15	22s	2
strk 106	No	66	2.2h	Yes	13	3s	No	50	625s	No	50	15m	No	88	15m	No	33	15m	2
strk 108	No	231	46m	No	65	20m	No	125	98s	No	261	47m	No	85	79s	No	92	162s	0
strk 111	Yes	66	7.5h	Yes	4	0	Yes	27	19m	Yes	23	72s	Yes	19	9s	Yes	15	2s	2
strk 112	No	230	35m	No	93	120s	No	252	35m	No	247	36m	No	82	131s	No	120	312s	0
strk 115	No	120	2.4h	Yes	21	68s	No	102	65s	No	106	122s	No	74	111s	No	63	15m	2
strk 116	No	130	2.4h	Yes	14	0	No	78	23s	No	40	15m	No	58	253s	No	112	90s	2
strk 118	No	240	377s	No	54	84s	No	262	379s	No	39	33s	No	70	130s	No	85	46m	0
strk 121	No	190	41m	Yes	38	8.5h	No	94	51s	No	79	59s	No	81	103s	No	78	93s	2
Total	**31**			**47**			**34**			**33**			**35**			**41**			**47**

h = hours; m = minute; s = second.

Appendix B: Two Sample Runs of a Canonical Semantic Tree Generator

```
Enter name of theorem (type '?' for help): stark026

Predicates: S P   Functions: A E G F
 1: [UM-] PxyFxy
 2: [UM-] PxGxE
 3: [UM-] PxEx
 4: [-M-] ~Pxys ~Pyuv ~Pxvw Psuw
 5: [-M-] ~Pxys ~Pyuv ~Psuw Pxvw
 6: [UM-] PExx
 7: [UM-] PGxxE
 8: [-M-] ~Sx ~Sy Ss ~PxGys
 9: [UM-] SA
10S [UM-] ~SGA
  1: SA                2: PAAA              3: SE
  4: PAAE              5: PAEA              6: PAEE
  7: PEAA              8: PEAE              9: PEEA
 10: PEEE             11: SGA              12: PAAGA
 13: PAEGA            14: PAGAA            15: PAGAE
 16: PAGAGA           17: PEAGA            18: PEEGA
 19: PEGAA            20: PEGAE            21: PEGAGA
 22: PGAAA            23: PGAAE            24: PGAAGA
 25: PGAEA

Press enter to continue.
Phase 1

Search dump:

Depth 1

NodesInTree = 2
 1
Depth 2

NodesInTree = 5
 1
Depth 3

NodesInTree = 9
 1
Depth 4

NodesInTree = 14
 1
Depth 5

NodesInTree = 21
 1
Depth 6

NodesInTree = 29
 1
Depth 7

NodesInTree = 39
 1
Depth 8

NodesInTree = 50
 1
Depth 9

NodesInTree = 63
 1
Depth 10

NodesInTree = 77
 1
Depth 11

NodesInTree = 92
 1
Depth 12

NodesInTree = 109
 1
Depth 13

NodesInTree = 128
 2
Depth 14

NodesInTree = 147
 2
Depth 15

NodesInTree = 233
 3
Depth 16

NodesInTree = 342
 4
Depth 17

NodesInTree = 441
 5
Depth 18

NodesInTree = 565
 7
Depth 19

NodesInTree = 660
 8
Depth 20

NodesInTree = 787
 10
Depth 21

PROOF FOUND!!!!!
NodesInTree = 1165
 15*

Given axioms:
 1: PxyFxy
 2: PxGxE
 3: PxEx
 4: ~Pxys ~Pyuv ~Pxvw Psuw
 5: ~Pxys ~Pyuv ~Psuw Pxvw
 6: PExx
 7: PGxxE
 8: ~Sx ~Sy Ss ~PxGys
 9: SA.

Negated conclusion:
10S ~SGA

Herbrand base atoms helped proving the theorem:
 1: SA
 2: ~PAAA
 3: SE
 4: ~PAAE
 5: PAEA
 6: ~PAEE
 7: PEAA
 8: ~PEAE
 9: ~PEEA
10: PEEE
11: ~SGA
12: PAAGA
13: ~PAEGA
14: ~PAGAA
15: PAGAE
16: ~PAGAGA
17: ~PEAGA
18: ~PEEGA
19: ~PEGAA
20: ~PEGAE
21: PEGAGA

ELM: 0 sec, PHS 1: 15 sec, PHS 2: -NA- Total Search Time: 15 sec
NOD: 1165     RES: 18154     FAC: 0      MXC: 7  MXL: 3
HTE: 0        HTH: 0         HTF: 0      HSZ: 262144
LTE: 0        LTH: 0         LTF: 0      LSZ: 65536
BAS: 10       LEN: 0 + 0 OPT: -SOS +MERGE


HBElems = 21

NodesInTree = 1165
```

The following example shows the canonical semantic tree generator printing the semantic tree on screen as being generated for S36wos12:

```
Enter name of theorem (type '?' for help): S36wos12.thm

Predicates: o r p   Functions: a e g f
 1: [UM-] pexx
 2: [UM-] pgxxe
 3: [UM-] pxyfxy
 4: [-M-] ~pxys ~pyuv ~psuw pxvw
 5: [-M-] ~pxys ~pyuv ~pxvw psuw
 6: [UM-] rxx
 7: [-M-] ~rxy ryx
 8: [-M-] ~rxy ~ryz rxz
 9: [-M-] rxu ~pxys ~pxyu
10: [-M-] ~rxy ~psux psuy
11: [-M-] ~rxy ~pszu pszu
12: [-M-] ~rxy ~pszu pyzu
13: [-M-] ~rxy rfxxfxy
14: [-M-] ~rxy rfxxfyx
15: [-M-] ~rxy rgxgy
16: [UM-] pxex
17: [UM-] pxgxe
18: [-M-] ~ox ~oy os ~pxgyx
19: [-M-] ~ox oy ~rxy
20: [UM-] oa
21S [UM-] ~oe
  1: oa                2: raa               3: paaa
  4: oe                5: rae               6: rea
  7: ree               8: paae              9: paea
 10: paee             11: peaa             12: peae
 13: peea             14: peee             15: oga
 16: raga             17: rega             18: rgaa
 19: rgae             20: rgaga            21: paaga
 22: paega            23: pagaa            24: pagae
 25: pagaga           26: peaga            27: peega
 28: pegaa            29: pegae            30: pegaga

Press enter to continue.
Phase 1

Search dump:

Depth 1


E FAILS!!!
E 1
E
E 2FAIL - MAX DEPTH
NodesInTree = 2
 0
Depth 2


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22FAIL - MAX DEPTH
NodesInTree = 6
 0
Depth 3


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAIL - MAX DEPTH
NodesInTree = 11
 0
Depth 4


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221
E 2211FAIL - MAX DEPTH
NodesInTree = 17
 0
Depth 5


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212
E 22121FAIL - MAX DEPTH
NodesInTree = 25
 0
Depth 6


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122
E 221221FAIL - MAX DEPTH
NodesInTree = 35
 0
Depth 7


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222FAIL - MAX DEPTH
NodesInTree = 45
 0
Depth 8


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAIL - MAX DEPTH
NodesInTree = 62
 0
Depth 9
```

```
E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAIL - MAX DEPTH
NodesInTree = 78
 0
Depth 10


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212
E 2212222121 FAIL - MAX DEPTH
NodesInTree = 95
 0
Depth 11


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAIL - MAX DEPTH
NodesInTree = 115
 0
Depth 12


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2
E 2212222122 21FAIL - MAX DEPTH
NodesInTree = 136
 0
Depth 13


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22
E 2212222122 221FAIL - MAX DEPTH
NodesInTree = 159
 0
Depth 14


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
```

```
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22
E 2212222122 221FAILS!!!
E 2212222122 2211
E 2212222122 221FAILS!!!
E 2212222122 2212
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222FAIL - MAX DEPTH
NodesInTree = 187
 1
Depth 15


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22
E 2212222122 221FAILS!!!
E 2212222122 2211
E 2212222122 221FAILS!!!
E 2212222122 2212
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2213222122 22221FAIL - MAX DEPTH
NodesInTree = 214
 1
Depth 16


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22FAILS!!!
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221FAILS!!!
E 2212222122 222211
E 2212222122 22221
E 2212222122 222212
E 2212222122 2222121FAIL - MAX DEPTH
NodesInTree = 244
 1
Depth 17


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22FAILS!!!
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221FAILS!!!
E 2212222122 222211
E 2212222122 22221
E 2212222122 222212
E 2212222122 2222121FAIL - MAX DEPTH
NodesInTree = 274
 1
Depth 18


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
```

```
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22
E 2212222122 221FAILS!!!
E 2212222122 2211
E 2212222122 221FAILS!!!
E 2212222122 2212
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221
E 2212222122 222211
E 2212222122 2222111FAILS!!!
E 2212222122 22221111
E 2212222122 2222111FAILS!!!
E 2212222122 22221112
E 2212222122 2222111
E 2212222122 222211
E 2212222122 2222112FAILS!!!
E 2212222122 22221121
E 2212222122 2222112
E 2212222122 22221122FAIL - MAX DEPTH
NodesInTree = 310
 1
Depth 19


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22FAILS!!!
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222FAILS!!!
E 2212222122 22221
E 2212222122 222211
E 2212222122 2222111
E 2212222122 22221111
E 2212222122 222211111FAILS!!!
E 2212222122 2222111111
E 2212222122 222211111
E 2212222122 2222111112 FAIL - MAX DEPTH
NodesInTree = 377
 2
Depth 21


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
```

```
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22FAILS!!!
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221
E 2212222122 222211
E 2212222122 2222111
E 2212222122 22221111
E 2212222122 222211111FAILS!!!
E 2212222122 2222111111
E 2212222122 222211111
E 2212222122 2222111112 FAILS!!!
E 2212222122 2222111112 1
E 2212222122 2222111112
E 2212222122 2222111112 2FAIL - MAX DEPTH
NodesInTree = 412
 2
Depth 22


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22FAILS!!!
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221
E 2212222122 222211
E 2212222122 2222111
E 2212222122 22221111
E 2212222122 222211111FAILS!!!
E 2212222122 2222111111
E 2212222122 222211111
E 2212222122 2222111112 FAILS!!!
E 2212222122 2222111112 1
E 2212222122 2222111112
E 2212222122 2222111112 2
E 2212222122 2222111112 21FAILS!!!
E 2212222122 2222111112 211
E 2212222122 2222111112 21
E 2212222122 2222111112 212FAIL - MAX DEPTH
NodesInTree = 469
 3
Depth 24


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
E 2212222122 2
E 2212222122 22FAILS!!!
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221
E 2212222122 222211
E 2212222122 2222111
E 2212222122 22221111
E 2212222122 222211111FAILS!!!
E 2212222122 2222111111
E 2212222122 222211111
E 2212222122 2222111112 FAILS!!!
E 2212222122 2222111112 1
E 2212222122 2222111112
E 2212222122 2222111112 2
E 2212222122 2222111112 21FAILS!!!
E 2212222122 2222111112 211
E 2212222122 2222111112 21
E 2212222122 2222111112 212FAIL - MAX DEPTH
NodesInTree = 469
 3
Depth 24


E FAILS!!!
E 1
E
E 2FAILS!!!
E 21
E 2
E 22
E 221FAILS!!!
E 2211
E 221
E 2212FAILS!!!
E 22121
E 2212
E 22122FAILS!!!
E 221221
E 22122
E 221222FAILS!!!
E 2212221
E 221222
E 2212222
E 22122221FAILS!!!
E 221222211
E 22122221
E 221222212FAILS!!!
E 2212222121
E 221222212
E 2212222122 FAILS!!!
E 2212222122 1
E 2212222122
E 2212222122 2FAILS!!!
E 2212222122 21
```

E 2212222122 2
E 2212222122 22
E 2212222122 221FAILS!!!
E 2212222122 2211
E 2212222122 221FAILS!!!
E 2212222122 2212
E 2212222122 221
E 2212222122 22
E 2212222122 222FAILS!!!
E 2212222122 2221
E 2212222122 222
E 2212222122 2222
E 2212222122 22221
E 2212222122 222211
E 2212222122 2222111
E 2212222122 22221111
E 2212222122 222211111FAILS!!!
E 2212222122 2222111111
E 2212222122 222211111
E 2212222122 2222111112 FAILS!!!
E 2212222122 2222111112 1
E 2212222122 2222111112
E 2212222122 2222111112 2
E 2212222122 2222111112 21FAILS!!!
E 2212222122 2222111112 211
E 2212222122 2222111112 21
E 2212222122 2222111112 212FAILS!!!
E 2212222122 2222111112 2121
E 2212222122 2222111112 212FAILS!!!
E 2212222122 2222111112 2122
E 2212222122 2222111112 212
E 2212222122 2222111112 21
E 2212222122 2222111112 2FAILS!!!
E 2212222122 2222111112 22
E 2212222122 2222111112 2
E 2212222122 222211112
E 2212222122 222211111
E 2212222122 22221111FAILS!!!
E 2212222122 222211112
E 2212222122 22221111
E 2212222122 2222111FAILS!!!
E 2212222122 22221112
E 2212222122 2222111
E 2212222122 222211
E 2212222122 2222112FAILS!!!
E 2212222122 22221121
E 2212222122 2222112FAILS!!!
E 2212222122 22221122
E 2212222122 2222112
E 2212222122 222211
E 2212222122 22221
E 2212222122 222212
E 2212222122 2222121FAILS!!!
E 2212222122 22221211
E 2212222122 2222121
E 2212222122 22221212
E 2212222122 222212121FAILS!!!
E 2212222122 2222121211
E 2212222122 222212121
E 2212222122 2222121212
E 2212222122 2222121212 1FAILS!!!
E 2212222122 2222121212 11
E 2212222122 2222121212 1
E 2212222122 2222121212 12
E 2212222122 2222121212 121FAILS!!!
E 2212222122 2222121212 1211
E 2212222122 2222121212 121FAILS!!!
E 2212222122 2222121212 1212
E 2212222122 2222121212 121
E 2212222122 2222121212 12
E 2212222122 2222121212 122FAILS!!!
E 2212222122 2222121212 1221
E 2212222122 2222121212 122FAILS!!!
E 2212222122 2222121212 1222
E 2212222122 2222121212 122
E 2212222122 2222121212 12
E 2212222122 2222121212 1
E 2212222122 2222121212 FAILS!!!
E 2212222122 2222121212 2
E 2212222122 2222121212
E 2212222122 222212121
E 2212222122 22221212FAILS!!!
E 2212222122 222212122
E 2212222122 22221213
E 2212222122 2222121
E 2212222122 222212
E 2212222122 2222122FAILS!!!
E 2212222122 22221221
E 2212222122 2222122
E 2212222122 22221222FAILS!!!
E 2212222122 222212221
E 2212222122 22221222FAILS!!!
E 2212222122 222212222
E 2212222122 22221222
E 2212222122 2222122
E 2212222122 222212
E 2212222122 22221
E 2212222122 2222
E 2212222122 222
E 2212222122 22
E 2212222122 2
E 2212222122
E 221222213
E 22122221
E 2212222FAILS!!!
E 221222223
E 22122222
E 2212222
E 221222
E 22122
E 2212
E 221
E 22
E 222FAILS!!!
E 2221

E 222
E 2222FAILS!!!
E 22221
E 2222
E 22222FAILS!!!
E 222221
E 22222
E 222222FAILS!!!
E 2222221
E 222222
E 2222222
E 22222221FAILS!!!
E 222222211
E 22222221
E 222222212FAILS!!!
E 2222222121
E 222222212
E 2222222122 FAILS!!!
E 2222222122 1
E 2222222122
E 2222222122 2FAILS!!!
E 2222222122 21
E 2222222122 2
E 2222222122 22FAILS!!!
E 2222222122 221
E 2222222122 22
E 2222222122 222FAILS!!!
E 2222222122 2221
E 2222222122 222
E 2222222122 2222
E 2222222122 22221
E 2222222122 222211
E 2222222122 2222111
E 2222222122 22221111
E 2222222122 222211111FAILS!!!
E 2222222122 2222111111
E 2222222122 2222111111
E 2222222122 2222111112
E 2222222122 2222111112 1
E 2222222122 2222111112 11
E 2222222122 2222111112 111FAILS!!!
E 2222222122 2222111112 1111
E 2222222122 2222111112 111FAILS!!!
E 2222222122 2222111112 1112
E 2222222122 2222111112 111
E 2222222122 2222111112 11FAILS!!!
E 2222222122 2222111112 112
E 2222222122 2222111112 11
E 2222222122 2222111112 1FAILS!!!
E 2222222122 2222111112 12
E 2222222122 2222111112 1
E 2222222122 2222111112 FAILS!!!
E 2222222122 2222111112 2
E 2222222122 2222111112
E 2222222122 222211111
E 2222222122 22221111FAILS!!!
E 2222222122 222211112
E 2222222122 22221111
E 2222222122 2222111FAILS!!!
E 2222222122 22221112
E 2222222122 2222111
E 2222222122 222211
E 2222222122 2222112FAILS!!!
E 2222222122 22221121
E 2222222122 2222112FAILS!!!
E 2222222122 22221122
E 2222222122 2222112
E 2222222122 222211
E 2222222122 22221
E 2222222122 222212
E 2222222122 2222121FAILS!!!
E 2222222122 22221211
E 2222222122 2222121211
E 2222222122 222212121
E 2222222122 2222121211
E 2222222122 2222121212
E 2222222122 2222121212 1FAILS!!!
E 2222222122 2222121212 11
E 2222222122 2222121212 1
E 2222222122 2222121212 12
E 2222222122 2222121212 121FAILS!!!
E 2222222122 2222121212 1211
E 2222222122 2222121212 121FAILS!!!
E 2222222122 2222121212 1212
E 2222222122 2222121212 121
E 2222222122 2222121212 12
E 2222222122 2222121212 122FAILS!!!
E 2222222122 2222121212 1221
E 2222222122 2222121212 122FAILS!!!
E 2222222122 2222121212 1222
E 2222222122 2222121212 122
E 2222222122 2222121212 12
E 2222222122 2222121212 1
E 2222222122 2222121212
E 2222222122 2222121212 2FAILS!!!
E 2222222122 2222121212 21
E 2222222122 2222121212 2
E 2222222122 2222121212 22
E 2222222122 2222121212 221FAILS!!!
E 2222222122 2222121212 2211
E 2222222122 2222121212 221FAILS!!!
E 2222222122 2222121212 2212
E 2222222122 2222121212 221
E 2222222122 2222121212 22
E 2222222122 2222121212 222FAILS!!!
E 2222222122 2222121212 2221
E 2222222122 2222121212 222FAILS!!!
E 2222222122 2222121212 2222
E 2222222122 2222121212 222
E 2222222122 2222121212 22
E 2222222122 2222121212 2
E 2222222122 2222121212
E 2222222122 222212121
E 2222222122 22221212FAILS!!!
E 2222222122 222212122
E 2222222122 22221213
E 2222222122 2222121
E 2222222122 222212
E 2222222122 2222122FAILS!!!
E 2222222122 22221221
E 2222222122 2222122
E 2222222122 22221222FAILS!!!
E 2222222122 222212221
E 2222222122 22221222FAILS!!!
E 2222222122 222212222
E 2222222122 22221222
E 2222222122 2222122
E 2222222122 222212
E 2222222122 22221
E 2222222122 2222
E 2222222122 22222FAILS!!!
E 2222222122 222221
E 2222222122 22222
E 2222222122 222222
E 2222222122 2222221FAILS!!!
E 2222222122 22222211
E 2222222122 2222221
E 2222222122 22222212
E 2222222122 222222121FAILS!!!
E 2222222122 2222221211
E 2222222122 222222121
E 2222222122 2222221212
E 2222222122 2222221212 1FAILS!!!
E 2222222122 2222221212 11
E 2222222122 2222221212 1
E 2222222122 2222221212 12
E 2222222122 2222221212 121FAILS!!!
E 2222222122 2222221212 1211
E 2222222122 2222221212 121FAILS!!!
E 2222222122 2222221212 1212
E 2222222122 2222221212 121
E 2222222122 2222221212 12
E 2222222122 2222221212 122FAILS!!!
E 2222222122 2222221212 1221
E 2222222122 2222221212 122FAILS!!!
E 2222222122 2222221212 1222
E 2222222122 2222221212 122
E 2222222122 2222221212 12
E 2222222122 2222221212 1
E 2222222122 2222221212
E 2222222122 2222221212 2FAILS!!!
E 2222222122 2222221212 21
E 2222222122 2222221212 2
E 2222222122 2222221212 22
E 2222222122 2222221212 221FAILS!!!
E 2222222122 2222221212 2211
E 2222222122 2222221212 221FAILS!!!
E 2222222122 2222221212 2212
E 2222222122 2222221212 221
E 2222222122 2222221212 22
E 2222222122 2222221212 222FAILS!!!
E 2222222122 2222221212 2221
E 2222222122 2222221212 222FAILS!!!
E 2222222122 2222221212 2222
E 2222222122 2222221212 222
E 2222222122 2222221212 22
E 2222222122 2222221212 2
E 2222222122 2222221212
E 2222222122 22222213
E 2222222122 222222132
E 2222222122 22222212FAILS!!!
E 2222222122 222222132
E 2222222122 22222312

E 2222222122 2222221
E 2222222122 222222
E 2222222122 2222222FAILS!!!
E 2222222122 22222221
E 2222222122 2222222
E 2222222122 22222222FAILS!!!
E 2222222122 222222221
E 2222222122 22222222
E 2222222122 222222222FAILS!!!
E 2222222122 2222222221
E 2222222122 222222222
E 2222222122 2222222222
E 2222222122 2222222222 1FAILS!!!
E 2222222122 2222222222 11
E 2222222122 2222222222 1
E 2222222122 2222222222 12FAILS!!!
E 2222222122 2222222222 121
E 2222222122 2222222222 12
E 2222222122 2222222222 122FAILS!!!
E 2222222122 2222222222 1221
E 2222222122 2222222222 122FAILS!!!
E 2222222122 2222222222 1222
E 2222222122 2222222222 122
E 2222222122 2222222222 12
E 2222222122 2222222222 1
E 2222222122 2222222222 FAILS!!!
E 2222222122 2222222222 2
E 2222222122 2222222222
E 2222222122 222222222
E 2222222122 22222222
E 2222222122 2222222
E 2222222122 222222
E 2222222122 22222
E 2222222122 2222
E 2222222122 222
E 2222222122 22
E 2222222122 2
E 2222222122
E 222222212
E 22222221
E 2222222
E 22222222FAILS!!!
E 222222221
E 22222222
E 2222222222 FAILS!!!
E 2222222222 1
E 2222222222
E 2222222222 2FAILS!!!
E 2222222222 21
E 2222222222 2
E 2222222222 22FAILS!!!
E 2222222222 221
E 2222222222 22
E 2222222222 222FAILS!!!
E 2222222222 2221
E 2222222222 222
E 2222222222 2222
E 2222222222 22221
E 2222222222 222211
E 2222222222 2222111
E 2222222222 22221111
E 2222222222 222211111FAILS!!!
E 2222222222 2222111111
E 2222222222 222211111
E 2222222222 2222111112
E 2222222222 2222111112 1
E 2222222222 2222111112 11
E 2222222222 2222111112 111FAILS!!!
E 2222222222 2222111112 1111
E 2222222222 2222111112 111FAILS!!!
E 2222222222 2222111112 1112
E 2222222222 2222111112 111
E 2222222222 2222111112 11FAILS!!!
E 2222222222 2222111112 112
E 2222222222 2222111112 11
E 2222222222 2222111112 1FAILS!!!
E 2222222222 2222111112 12
E 2222222222 2222111112 1
E 2222222222 2222111112 FAILS!!!
E 2222222222 2222111112 2
E 2222222222 2222111112
E 2222222222 222211111
E 2222222222 22221111FAILS!!!
E 2222222222 222211112
E 2222222222 22221111
E 2222222222 2222111
E 2222222222 222211

E 2222222222 2222112FAILS!!!
E 2222222222 22221121
E 2222222222 2222112FAILS!!!
E 2222222222 22221122
E 2222222222 2222112
E 2222222222 222211
E 2222222222 22221
E 2222222222 222212
E 2222222222 2222121FAILS!!!
E 2222222222 22221211
E 2222222222 2222121
E 2222222222 22221212
E 2222222222 222212121FAILS!!!
E 2222222222 2222121211
E 2222222222 222212121
E 2222222222 2222121212
E 2222222222 2222121212 1FAILS!!!
E 2222222222 2222121212 11
E 2222222222 2222121212 1
E 2222222222 2222121212 12
E 2222222222 2222121212 121FAILS!!!
E 2222222222 2222121212 1211
E 2222222222 2222121212 121FAILS!!!
E 2222222222 2222121212 1212
E 2222222222 2222121212 121
E 2222222222 2222121212 12
E 2222222222 2222121212 122FAILS!!!
E 2222222222 2222121212 1221
E 2222222222 2222121212 122FAILS!!!
E 2222222222 2222121212 1222
E 2222222222 2222121212 122
E 2222222222 2222121212 12
E 2222222222 2222121212 1
E 2222222222 2222121212 FAILS!!!
E 2222222222 2222121212 2
E 2222222222 2222121212
E 2222222222 222212121
E 2222222222 22221212FAILS!!!
E 2222222222 2222121212
E 2222222222 222212121
E 2222222222 22221212
E 2222222222 2222121
E 2222222222 222212
E 2222222222 2222122FAILS!!!
E 2222222222 22221221
E 2222222222 2222122
E 2222222222 22221222FAILS!!!
E 2222222222 222212221
E 2222222222 22221222FAILS!!!
E 2222222222 222212222
E 2222222222 22221222
E 2222222222 2222122
E 2222222222 222212
E 2222222222 22221
E 2222222222 2222
E 2222222222 22222FAILS!!!
E 2222222222 222221
E 2222222222 22222
E 2222222222 222222
E 2222222222 2222221FAILS!!!
E 2222222222 22222211
E 2222222222 2222221
E 2222222222 22222212
E 2222222222 222222121FAILS!!!
E 2222222222 2222221211
E 2222222222 222222121
E 2222222222 2222221212
E 2222222222 2222221212 1FAILS!!!
E 2222222222 2222221212 11
E 2222222222 2222221212 1
E 2222222222 2222221212 12
E 2222222222 2222221212 121FAILS!!!
E 2222222222 2222221212 1211
E 2222222222 2222221212 121FAILS!!!
E 2222222222 2222221212 1212
E 2222222222 2222221212 121
E 2222222222 2222221212 12
E 2222222222 2222221212 122FAILS!!!
E 2222222222 2222221212 1221
E 2222222222 2222221212 122FAILS!!!
E 2222222222 2222221212 1222
E 2222222222 2222221212 122
E 2222222222 2222221212 12
E 2222222222 2222221212 1
E 2222222222 2222221212 FAILS!!!
E 2222222222 2222221212 2
E 2222222222 2222221212
E 2222222222 222222121
E 2222222222 22222212FAILS!!!
E 2222222222 222222122
E 2222222222 22222212
E 2222222222 2222221
E 2222222222 222222

```
E 2222222222 2222222FAILS!!!
E 2222222222 22222221
E 2222222222 2222222
E 2222222222 22222222FAILS!!!
E 2222222222 22222221
E 2222222222 22222222
E 2222222222 222222222FAILS!!!
E 2222222222 2222222221
E 2222222222 2222222222
E 2222222222 2222222222 FAILS!!!
E 2222222222 2222222222 1
E 2222222222 2222222222
E 2222222222 2222222222 2FAILS!!!
E 2222222222 2222222222 21
E 2222222222 2222222222 2
E 2222222222 2222222222 22FAILS!!!
E 2222222222 2222222222 221
E 2222222222 2222222222 22
E 2222222222 2222222222 222FAILS!!!
E 2222222222 2222222222 2221
E 2222222222 2222222222 222FAILS!!!
E 2222222222 2222222222 2222
E 2222222222 2222222222 222
E 2222222222 2222222222 22
E 2222222222 2222222222 2
E 2222222222 2222222222
E 2222222222 222222222
E 2222222222 22222222
E 2222222222 2222222
E 2222222222 222222
E 2222222222 22222
E 2222222222 2222
E 2222222222 222
E 2222222222 22
E 2222222222 2
E 2222222222
E 222222222
E 22222222
E 2222222
E 222222
E 22222
E 2222
E 222
E 22
E 2
E
E PROOF FOUND!!!!!
NodesInTree == 803
S*

Given axioms:
 1: pexx
 2: pgxxe
 3: pxyfxy
 4: ~pxys ~pyxv ~pxxw pxvw
 5: ~pxys ~pyxv ~pxvw pxxw
 6: rxx
 7: ~rxy ryx
 8: ~rxy ~ryz rxz
 9: rxu ~pxyz ~pxyu
10: ~rxy ~pxux pxuy
11: ~rxy ~pxxu pxyu
12: ~rxy ~pxxu pyxu
13: ~rxy rfxxfxy
14: ~rxy rfxxfyx
15: ~rxy rgxgy
16: pxex
17: pxgxe
18: ~ox ~oy os ~pxgys
19: ~ox oy ~rxy
20: oa

Negated conclusion:
21S ~oe

Herbrand base atoms helped proving the theorem:
 1: oa
 2: rxa
 3: ~pxxa
 4: ~oe
 5: ~rxe
 6: ~rea
 7: ree
 8: pxxe
 9: pexa
10: ~pxee
11: pexa
12: ~pexe
13: ~pexa
14: pexe
15: ~oga
16: ~rega
17: rega
18: ~rgaa
19: rgae
20: rgaga
21: paaga
22: ~paega
23: pagaa
24: pgae

ELM: 0 sec, PHS 1: 8 sec, PHS 2: -NA-  Total Search Time: 8 sec
NOD: 803      RES: 9901      FAC: 0      MXC: 8  MXL: 3
HTE: 0        HTH: 0         HTF: 0      HSZ: 262144
LTE: 0        LTH: 0         LTF: 0      LSZ: 65536
BAS: 21       LEN: 0 + 0 OPT: -SOS +MERGE

HBElems = 24

NodesInTree = 803
```

Appendix C: Sample Runs of An Improved Semantic Tree Generator

```
Enter name of theorem (type '?' for help): stark068

Predicates: E L   Functions: S 0 P M F A
 1: [-M-] ~ESxSy Exy
 2: [UM-] ~ESx0
 3: [-M-] ~Exy ~Exz Eyz
 4: [-M-] ~Exy ESxSy
 5: [UM-] EPx0x
 6: [UM-] EPxSySPxy
 7: [UM-] EMx00
 8: [UM-] EMxSyPMxyx
 9: [UM-] Exx
10: [-M-] ~Exy Eyx
11: [-M-] ~Exy ~Eyz Exz
12: [-M-] EPSFxyxy ~Lxy
13: [-M-] ~EPSxyz Lyz
14: [UM-] LxSx
15S [UM-] ~L0SA
Phase 1

Search dump:

==============================
—      Choosing a Strategy    —
==============================
1 - Unit Resolution (UR).
2 - Fewest Literals Strategy (FLS).
3 - Set Of Support (SOS).
4 - Linear Form (LF).
5 - FLS + SOS.
6 - FLS + LF.
7 - Exit Program.


Enter strategy number : 3

Depth 1

search'tree'SOS()

E FAILS!!!
E 1
E
E 2FAIL - MAX DEPTH
 NEW BASE (2)?? 1: (15..)    ~L0SA
 2: (16..)    EPSF0SA0SA
NodesInTree = 2
 0
Depth 2

search'tree'SOS()

E FAILS!!!
E 1
E
E 2
E 21FAIL - MAX DEPTH
 NEW BASE (4)?? 1: (15..)    ~L0SA
 2: (16..)    ~EPSF0SA0SA
 3: (16..)    EPSA0SA
 4: (16..)    EPS00SA
NodesInTree = 5
 0
Depth 3

search'tree'SOS()

E FAILS!!!
E 1
E
E 2
E 21FAILS!!!
E 211
E 21FAILS!!!
E 212
E 21
E 2FAILS!!!
E 22
E 2
E
E PROOF FOUND!!!!!
NodesInTree = 11
 0*

Given axioms:
 1: ~ESxSy Exy
 2: ~ESx0
 3: ~Exy ~Exz Eyz
 4: ~Exy ESxSy
 5: EPx0x
 6: EPxSySPxy
 7: EMx00
 8: EMxSyPMxyx
 9: Exx
10: ~Exy Eyx
11: ~Exy ~Eyz Exz
12: EPSFxyxy ~Lxy
13: ~EPSxyz Lyz
14: LxSx

Negated conclusion:
15S ~L0SA

Herbrand base atoms helped proving the theorem:
 1: ~L0SA              2: EPSF0SA0SA          3: EPSA0SA
 4: EPS00SA


ELM: 0 sec, PHS 1: 0 sec, PHS 2: -NA- Total Search Time: 0 sec
NOD: 11       RES: 30       FAC: 0        MXC: 13  MXL: 8
HTE: 0        HTH: 0        HTF: 0        HSZ: 262144
LTE: 0        LTH: 0        LTF: 0        LSZ: 65536
BAS: 15       LEN: 0 + 0 OPT: -SOS +MERGE

**************************************************************************
**************************************************************************
**************************************************************************

   The following example shows the AISTG prove Starkey100 using the
Set-Of-Support strategy.


Enter name of theorem (type '?' for help): stark100

Predicates: S M E   Functions: F b c a
 1: [-M-] ~Sxy ~Msx Msy
 2: [-M-] Sxy MFxyx
 3: [-M-] Sxy ~MFxyy
 4: [-M-] Sxy ~Exy
 5: [-M-] Sxy ~Eyx
 6: [-M-] ~Sxy ~Syx Exy
 7: [UM-] Ebc
 8: [UM-] Mab
 9S [UM-] ~Mac
Phase 1

Search dump:

==============================
—      Choosing a Strategy    —
==============================
1 - Unit Resolution (UR).
2 - Fewest Literals Strategy (FLS).
3 - Set Of Support (SOS).
4 - Linear Form (LF).
5 - FLS + SOS.
6 - FLS + LF.
7 - Exit Program.


Enter strategy number : 3

Depth 1


E Checking:  1: (7..)     [U—] ~Ebc
FAILS!!!
E 1
E Checking:  1: (7..)     [U—] Ebc

E 2FAIL - MAX DEPTH
 NEW BASE (3)??
 1: (7..)     Ebc
 2: (8..)     Mab
 3: (9..)     ~Mac
NodesInTree = 2
 0
Depth 2


E Checking:  1: (7..)     [U—] ~Ebc
FAILS!!!
E 1
E Checking:  1: (7..)     [U—] Ebc

E 2Checking:  2: (8..)     [U—] ~Mab
FAILS!!!
E 21
E 2Checking:  2: (8..)     [U—] Mab

E 22FAIL - MAX DEPTH
 NEW BASE (3)??
```

```
 1: (7..)     Ebc
 2: (8..)     Mab
 3: (9..)     ~Mac
NodesInTree = 6
 0
Depth 3


E Checking:  1: (7..)     [U—] ~Ebc
FAILS!!!
E 1
E Checking:  1: (7..)     [U—] Ebc

E 2Checking:  2: (8..)     [U—] ~Mab
FAILS!!!
E 21
E 2Checking:  2: (8..)     [U—] Mab

E 22Checking:  3: (9..)     [U—] Mac
FAILS!!!
E 221
E 22Checking:  3: (9..)     [U—] ~Mac

E 222FAIL - MAX DEPTH
 NEW BASE (5)??
 1: (7..)     Ebc
 2: (8..)     Mab
 3: (9..)     ~Mac
 4: (10..)    Sbc
 5: (11..)    Scb
NodesInTree = 12
 0
Depth 4


E Checking:  1: (7..)     [U—] ~Ebc
FAILS!!!
E 1
E Checking:  1: (7..)     [U—] Ebc

E 2Checking:  2: (8..)     [U—] ~Mab
FAILS!!!
E 21
E 2Checking:  2: (8..)     [U—] Mab

E 22Checking:  3: (9..)     [U—] Mac
FAILS!!!
E 221
E 22Checking:  3: (9..)     [U—] ~Mac

E 222Checking:  4: (10..)    [U—] ~Sbc
FAILS!!!
E 2221
E 222Checking:  4: (10..)    [U—] Sbc
FAILS!!!
E 2222
E 222
E 22
E 2
E
E PROOF FOUND!!!!!
NodesInTree = 20
 0*

Given axioms:
 1: ~Sxy ~Msx Msy
 2: Sxy MFxyx
 3: Sxy ~MFxyy
 4: Sxy ~Exy
 5: Sxy ~Eyx
 6: ~Sxy ~Syz Exy
 7: Ebc
 8: Mab

Negated conclusion:
 9S ~Mac

Herbrand base atoms helped proving the theorem:
  1: Ebc                2: Mab                3: ~Mac
  4: Sbc                5: Scb


ELM: 0 sec, PHS 1: 0 sec, PHS 2: -NA- Total Search Time: 0 sec
NOD: 20       RES: 33       FAC: 0        MXC: 6  MXL: 8
HTE: 0        HTH: 0        HTF: 0        HSZ: 262144
LTE: 0        LTH: 0        LTF: 0        LSZ: 65536
BAS: 9        LEN: 0 + 0 OPT: -SOS +MERGE


HBElems = 4

NodesInTree = 20
```

```
**************************************************************
**************************************************************
**************************************************************
```

The following example shows the improved semantic tree generator printing a proof extracted from the closed semantic tree of S43wos19:

```
Enter name of theorem (type '?' for help): S43wos19.thm

Predicates: p r o   Functions: e g f i a b c d
 1: [UM-] pexx
 2: [UM-] pgxxe
 3: [UM-] pxyfxy
 4: [-M-] ~pxys ~pyzv ~pzuw pxvw
 5: [-M-] ~pxys ~pyzv ~pxvw pzuw
 6: [UM-] rxx
 7: [-M-] ~rxy ryx
 8: [-M-] ~rxy ~ryz rxz
 9: [-M-] ~pxys ~pxyu rzu
10: [-M-] ~pzux pzuy ~rxy
11: [-M-] ~pzxu pzyu ~rxy
12: [-M-] ~pzxu pyzu ~rxy
13: [-M-] ~rxy rfxxfsy
14: [-M-] ~rxy rfxxfys
15: [-M-] ~rxy rgxgy
16: [UM-] pxex
17: [UM-] pxgxe
18: [-M-] ~pxys ~ox ~oy os
19: [-M-] ~rxy ~ox oy
20: [-M-] ~ox ogx
21: [UM-] oe
22: [-M-] ~rxy rixxiy
23: [-M-] ~rxy rixxiys
24: [-M-] ox oy oixy
25: [-M-] pxixyy ox oy
26: [-M-] ~pxys ~pxzs ryz
27: [-M-] ~pxys ~pzys rzz
28: [UM-] rggxx
29: [UM-] oa
30: [UM-] ob
31: [UM-] pbgac
32: [UM-] pacd
338 [UM-] ~od

Search dump:

Depth 1

 0
Depth 2

 0
Depth 3

 2
Depth 4

 2
Depth 5

 3
Depth 6

 3
Depth 7

 3
Depth 8

 3
Depth 9

 4
Depth 10

 4*

Given axioms:
 1: pexx
 2: pgxxe
 3: pxyfxy
 4: ~pxys ~pyzv ~pzuw pxvw
 5: ~pxys ~pyzv ~pxvw pzuw
 6: rxx
 7: ~rxy ryx
 8: ~rxy ~ryz rxz
 9: ~pxys ~pxyu rzu
10: ~pzux pzuy ~rxy
11: ~pzxu pzyu ~rxy
```

```
12: ~pxsu pysu ~rxy
13: ~rxy rfsxfsy
14: ~rxy rfxsfys
15: ~rxy rgxgy
16: pxex
17: pxgxe
18: ~pxys ~ox ~oy os
19: ~rxy ~ox oy
20: ~ox ogx
21: oe
22: ~rxy risxisy
23: ~rxy rixsiys
24: ox oy oixy
25: pxixyy ox oy
26: ~pxys ~pxus ryu
27: ~pxys ~puys rxu
28: rggxx
29: oa
30: ob
31: pbgac
32: pacd

Negated conclusion:
33S ~od
 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
 5: 32..   pacd
 6: 33..   ~od
 7: 37..   oge
 8: 42..   oga
 9: 49..   ogb
10: 66..   oc
36: 18c,18d ~pxys ~puvy ~ox os ~ou ~ov

Depth 9

 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
 5: 32..   pacd
 6: 33..   ~od
 7: 38..   oge
 8: 50..   oga
 9: 76..   ogb
37: 34f,20b ~pxys ~pugvy ~ox os ~ou ~ov

Depth 8

 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
 5: 32..   pacd
 6: 33..   ~od
 7: 39..   oge
 8: 58..   oga
38: 35d,33a ~pxyd ~puguy ~ox ~os ~ou

Depth 7

 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
 5: 32..   pacd
 6: 33..   ~od
 7: 0..    oge
39: 36a,32a ~pxgyc ~oa ~ox ~oy

Depth 6

 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
 5: 32..   pacd
 6: 33..   ~od
40: 37a,31a ~oa ~ob

Depth 5

 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
 5: 32..   pacd
41: 38b,30a ~oa

Depth 4

 1: 21..   oe
 2: 29..   oa
 3: 30..   ob
 4: 31..   pbgac
42: 39a,29a []

 1: pexx
 2: pgxxe
 3: pxyfxy
 4: ~pxys ~pyuv ~psuw pxvw
 5: ~pxys ~pyuv ~pxvw psuw
 6: rxx
 7: ~rxy ryx
 8: ~rxy ~rys rxs
 9: ~pxys ~pxys rsu
10: ~psux psuy ~rxy
11: ~psxs psys ~rxy
12: ~pxsu pysu ~rxy
13: ~rxy rfsxfsy
14: ~rxy rfxsfys
15: ~rxy rgxgy
16: pxex
17: pxgxe
18: ~pxys ~ox ~oy os
19: ~rxy ~ox oy
20: ~ox ogx
21: oe
22: ~rxy risxisy
23: ~rxy rixsiys
24: ox oy oixy
25: pxixyy ox oy
26: ~pxys ~pxus ryu
27: ~pxys ~puys rxu
28: rggxx
29: oa
30: ob
31: pbgac
32: pacd
33S ~od
34: 18c,18d ~pxys ~puvy ~ox os ~ou ~ov
35: 34f,20b ~pxys ~pugvy ~ox os ~ou ~ov
36: 35d,33a ~pxyd ~puguy ~ox ~os ~ou
37: 36a,32a ~pxgyc ~oa ~ox ~oy
38: 37a,31a ~oa ~ob
39: 38b,30a ~oa
40: 39a,29a []


ELIM: 0 sec, PHS 1: 4 sec, PHS 2: -NA- Total Search Time: 4 sec
NOD: 166      RES: 3800      FAC: 0      MXC: 8  MXL: 8
HTE: 0        HTH: 0         HTF: 0        HSZ: 262144
LTE: 0        LTH: 0         LTF: 0        LSZ: 65536
BAS: 33       LEN: 0 + 0 OPT: -SOS +MERGE


HBElems = 4

NodesInTree = 166
```