Semantically Guided First-Order Theorem Proving with Hyper-Linking

by

Heng Chu

A dissertation submitted to the faculty of the University of North Carolina in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill, 1994

Approved by:

Advisor: David A. Plaisted

Reader: Donald W. Loveland

Reader: Gyula A. Mago

HENG CHU

Semantically Guided First-Order Theorem Proving with Hyper-Linking

(Under the direction of David A. Plaisted)

ABSTRACT

An automated theorem prover accepts a set of axioms and a theorem, expressed in a formal logic, and then proves that the theorem logically follows from the axioms. Two basic aspects of a logic are *syntax* and *semantics*. Syntax refers to the form of the statements and semantics refers to their meanings. Most current theorem provers conduct proof searching based on syntax only and semantics is rarely used.

In this dissertation we propose a new and complete theorem proving method, called *semantic hyper-linking*, which makes a substantial use of semantics in an instance-based theorem prover for first-order logic. Semantics is used to generate instances by a combination of unification and enumeration of the Herbrand base. In this method, the proof searching is essentially a process of exhaustively searching for a model of the input problem.

Semantic hyper-linking performs especially well on non-Horn problems whose proofs contain small instances of the input clauses. We also describe several refinements including (1) various model finding strategies to improve model searching without using expensive semantic checks or expanding the search space, (2) the rough resolution method to cooperate with semantic hyper-linking and solve the part of the proof which contains large instances, and (3) the well-known UR resolution to solve the Horn part of the proof.

We have implemented semantic hyper-linking with its refinements, and applied it to over a hundred problems in various domains. Our results show that semantic hyper-linking is indeed a promising technique. Several hard theorems, which many other theorem provers find difficult, have been proved by semantic hyper-linking using no other guidance than the given semantics.

ACKNOWLEDGMENTS

First I would like to thank Professor David Plaisted for being an important source of guidance, suggestions and support throughout the duration of this research. I am very lucky to have him as my advisor, from whom I have learned how to appreciate the joy of conducting basic research.

I would also like to express my gratitude to the other members of my committee; Professor Gyula Magó, for giving me so much advice at the final stage of this research while Professor Plaisted was on sabbatical leave, especially during the preparation of the dissertation defense presentation; Professor Donald Loveland, for his suggestions and guidance as an expert in the theorem proving field, and for his willingness to travel to Chapel Hill from Duke University for the committee meetings; Dr. Susanna Schwab, for her persistent encouragement and advice since I was a master student working with her, and for her traveling from BNR to Chapel Hill for my dissertation defense despite her busy schedule; Professor Peter Calingaert, for his kind assistance and advice regarding the graduate school regulations; and Professor Donald Stanat, for his valuable advice about my dissertation and for providing immediate help on many occasions.

I am also grateful to my fellow graduate students James Riely, Muthukrishnan Paramasivam, Ta-Ming Chen and Yunshan Zhu for the valuable suggestions during my defense preparation; to Geoffrey Alexander for being such a wonderful officemate, with whom I have enjoyed the conversation and fruitful discussion all along; and to the Computer Science Department for providing me such a wonderful environment.

I would like to thank National Science Foundation for supporting this research under grant CCR-9108904.

I would also like to thank my parents for their everlasting caring love and all the things they have taught me; and my brothers, especially Dr. Jinn Chu, for all their support. Finally and most of all, I want to thank my lovely wife Ming-Ju. This dissertation could not have been done without the love and support from her, to whom I dedicate this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Automated Theorem Proving

Automated theorem proving is a subfield in machine reasoning of artificial intelligence (AI) devoted to machine reasoning and focuses on investigations of mechanized inference methods to prove that a theorem logically follows from a set of axioms. Although theorem proving research is of great value (and fun) by itself, there are also a wide range of practical applications of automated theorem proving including software and hardware verification which check that implementations meet the requirements or the specifications, inference engines in knowledge-based systems to reason about knowledge stored in the knowledge base, and other tasks such as answering puzzles and real-time systems control [WOLB84]. In particular, automated theorem provers also have been used to answer open questions in areas such as ternary Boolean theory, finite semi-groups, Robbins algebra, equivalential calculus, the R-calculus, and the L-calculus [WW84, Win82].

Formal logic (such as first-order logic to be discussed later) is often used to symbolize theorems and axioms. An automated theorem prover applies inference rules and automatically proves theorems expressed in a certain formal logic.

First-order logic is a formal logic widely used to represent problems or concepts. The representation can be in either first-order logic or clause form (a restricted first-order logic to be discussed in Chapter 2). In this dissertation, we will focus on first-order clause form.

There are several inference methods available for first-order clause form theorem proving. Popular methods include *resolution*, *model elimination*, *the connection method*, and *the instance-based method*. Here we will only review briefly the resolution method and the instance-based approach.

Because of its simplicity and efficiency, the resolution technique has been very popular in the theorem proving community and many refinements have been proposed since its introduction (described in Section 2.4.3). On the other hand, the instance-based (also called *Herbrand-based*) theorem proving method, although being a direct application of a major theorem for clause form theorem proving, was not efficient when it was first introduced and has not been popular until recently with the introduction of the hyper-linking method by Lee and Plaisted [LP92, Lee90, CLP94] (described in Section 2.6).

In general, there are two basic aspects of a logic: *syntax* and *semantics*. Syntax involves symbol manipulation which specifies the grammar of a formal language and its constituent parts. Syntax mainly concerns the logical expressions which can be legally constructed. In contrast, semantics assigns meanings to expressions of the formal language. When we solve mathematical problems, we usually have semantics for the problems and often construct a picture in our minds. For the same reason, we can use semantics to understand or interpret an expression in logic. The relationship between syntax and semantics of a logic is thus important.

The majority of current theorem proving techniques search for the proofs based on the syntax of the input problems. The basic operation is a syntactic matching process. We argue that semantics should also be used (through the use of an example for the input problem) and semantics can provide guidance in the search of a proof. Although there are previous research works investigating such an approach, none applies to instance-based clause form theorem proving. We say such a theorem prover is *semantically guided*.

Based on the hyper-linking method, a research problem naturally suggests itself: Can semantics be used with Herbrand-based theorem proving efficiently? One major advantage of using semantics with Herbrand-based theorem proving is that literals from different clauses are not combined so it is efficient to do semantic checks (a process that uses semantics to determine if a clause should be kept). This dissertation is devoted to answering such a question by introducing a new clause form theorem proving technique.

1.2 The Thesis Statement

In this dissertation, we give an affirmative answer to the aforementioned research question, and describe the use of semantics in a Herbrand-based theorem prover to help find faster proofs. The underlying instance-based method is hyper-linking which has been shown efficient and promising.

Our thesis is: *Semantics (examples) can be used effectively with hyper-linking to prove theorems efficiently.*

The basic method is *semantic hyper-linking* which uses the input example as semantics and systematically searches for a model for the input clauses. One important fact about semantic hyper-linking is that it is complete regardless of the given semantics (or example). Thus this method can also be used as a syntax-oriented method if the input semantics is trivial (for example, one in which true literals are exactly the positive ones).

We also develop several refinement procedures to help the basic method conduct a faster proof search. In particular, semantic hyper-linking is good for solving small non-Horn parts of a proof; rough resolution solves large parts of a proof which contain large instances; UR resolution is a popular method good for solving small Horn parts of a proof. It is shown in this dissertation that the combination of these methods is efficient and can prove several problems that other major proving techniques find difficult to prove.

1.3 Outline of the Thesis

In the next chapter, we will review major background materials on mathematical logic, theorem proving and related research. In Chapter 3, we present the basic method of semantic hyper-linking. First an example of model searching is given and then it is shown how model searching can be used in first-order theorem proving. Detailed discussions of the method are given, followed by an example using semantic hyper-linking to prove a theorem in group theory.

Chapter 4 further discusses several model finding strategies for a ground instance set of a set of first-order clauses. The main idea of those strategies is to find a better model that satisfies the input clause set and some of the logical consequences of the input clauses. We also discuss how UR resolution can be used with semantic hyper-linking as a major component for solving the "Horn part" of proofs. In particular, we discuss how to use two different criteria to order the application of UR resolution—one based on the total size of the resolvents; the other on the total proof size of the resolvents. In addition, we discuss how to balance these two types of UR resolution.

Chapter 5 presents a refinement of resolution, called rough resolution, to cope with a major problem of semantic hyper-linking, namely, that the generation of some large ground literals, if needed for the proof, is difficult. Rough resolution focuses on resolving on the literals whose ground literals are large so they are no longer needed if ground instances of resolvents are also generated.

In Chapter 6, the choice and use of semantics with decision procedures are discussed. In particular, we show how semantics in Bledsoe real arithmetic is used with our method. Expressions in Bledsoe real arithmetic have a decision procedure Sup-Inf. We discuss some issues in using Bledsoe real arithmetic; we also show how to "simulate" some semantics in other domains by Bledsoe real arithmetic.

Chapter 7 discusses the difference between the use of domain-specific knowledge and human control. Also we present a unified view of our methods.

In Chapter 8 we discuss the implementation of a prover, `CLIN-S`, using semantic hyper-linking and list some experimental results by running `CLIN-S` on some theorems that are difficult for many other commonly used provers.

Chapter 9 concludes the dissertation and discusses some directions of future work.

Chapter 2

Background

In this chapter we first briefly review the logic, *the propositional calculus* and *first-order logic*, that our methods are based upon. Then we describe theorem proving basics and major techniques. We also describe hyper-linking and related works in semantically guided theorem proving. Informal descriptions accompany formal definitions to give better intuitive understanding. We do not intend to cover all the basic materials; we review only those important to understand our work. For complete coverage, please refer to [CL73, Lov78, Pla90, WOLB84].

2.1 Propositional Calculus

Before we discuss the logic of our prover, we briefly introduce a simpler logic called the *propositional logic (calculus)*. Though simple, the propositional calculus plays a central role in our work. We present the discussion from syntactic and semantic aspects; we also discuss some related topics.

2.1.1 Syntax

We call a declarative sentence like "John plays guitar" which is either *true* or *false* (but not both), a *proposition*. We symbolize a proposition by an *atom*. For example, we can use the atom P to denote the proposition "John plays guitar." Propositional calculus focuses on the study of the behavior of the *logical connectives* which are used to build *compound* propositions like "John plays guitar *and* Mary plays piano."

In the following discussions, we use five common logical connectives as described in Table 2.1.

$\neg$	negation (*not*)
$\wedge$	conjunction (*and*)
$\vee$	disjunction (*or*)
$\supset$	implication (*if then*)
$\equiv$	equivalence (*if and only if*)

Table 2.1: Logical connectives

For example, let Q represent "Mary plays piano," then $(P \wedge Q)$ can represent the compound proposition in the previous example.

The following definitions specify the legitimate expression syntax in the propositional calculus.

Definition 1. *Well-formed formulas* (*wffs* or simply *formulas*) in the propositional calculus are defined as follows:

(1) An atom is a formula;

(2) If G is a formula, then $\neg G$ is a formula;

(3) If G and H are formulas, then $G \wedge H$, $G \vee H$, $G \supset H$, and $G \equiv H$ are all formulas;

(4) All formulas are generated by applying the above rules. □

Any formula can be transformed to a *normal form*. In particular, in our theorem proving method, we use clause form defined as follows.

Definition 2. A *literal* is an atom or the negation of an atom. Literals A and $\neg A$ are said to be each other's *complements*, and the set $\{A, \neg A\}$ is called a *complementary pair*. An atom is a *positive literal* and the negation of an atom is a *negative literal*.

Definition 3. An *atom set* S of a formula F is the set of all atoms for which the corresponding positive or negative literals appear in F, namely, an atom set of S is the following:

$$\{A | \text{either } A \text{ or } \neg A \text{ appears in } F\}$$

	$(F \equiv G) = (F \supset G) \wedge (G \supset F)$
	$(F \supset G) = (\neg F \vee G)$
Distributivity	(a) $(F \vee (G \wedge H)) = ((F \vee G) \wedge (F \vee H))$
	(b) $(F \wedge (G \vee H)) = ((F \wedge G) \vee (F \wedge H))$
	$\neg(\neg F) = F$
De Morgan's laws	(a) $\neg(F \vee G) = (\neg F \wedge \neg G)$
	(b) $\neg(F \wedge G) = (\neg F \vee \neg G)$

Table 2.2: Equivalent formulas

Definition 4. $F_1 \vee F_2 \vee \ldots \vee F_n$ (each F_i is a formula) is called the *disjunction* of $F_1, F_2, \ldots, F_n$. $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is called the *conjunction* of $F_1, F_2, \ldots, F_n$.

For a formula F in the propositional logic, we can apply the equivalences $(P \equiv Q) = ((P \supset Q) \wedge (Q \supset P))$ and $(P \supset Q) = (\neg P \vee Q)$ to remove all the $\equiv$ and $\supset$ in F, so that only connectives $\neg$, $\wedge$ and $\vee$ are retained. In particular, we can obtain the following normal forms by properly arranging logic connectives.

Definition 5. A *clause* is a disjunction of literals. A formula is in *clause form*, or *conjunctive normal form* (CNF) if it is a conjunction of clauses. A formula F is in *disjunctive normal form* (DNF) if it is $F_1 \vee F_2 \vee \ldots \vee F_n$ and each F_i is a conjunction of literals.

Every formula has an equivalent CNF and DNF which can be computed in exponential time by applying the equivalent relations shown in Table 2.2 where F, G and H are all formulas.

In later discussions, we always express a clause as a set of literals (disjunction of literals), and a conjunction of clauses as a set of clauses (conjunction of clauses).

2.1.2 Semantics

Each atom has a *truth value* which is either true (**T**) or false (**F**). Logic connectives give commonly known truth value evaluation in terms of the sub-formulas involved (see Table 2.3). For example, $\neg$**T** gives truth value **F** and **T** $\vee$ **F** gives truth value **T**. Thus the truth value of any formula can be evaluated in terms of truth values of atoms.

G	H	$\neg G$	$(G \wedge H)$	$(G \vee H)$	$(G \supset H)$	$(G \equiv H)$
T	**T**	**F**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**	**F**
F	**T**	**T**	**F**	**T**	**T**	**F**
F	**F**	**T**	**F**	**F**	**T**	**T**

Table 2.3: Truth assignments for the propositional calculus

Definition 6. Assignment of the truth values to the atoms in a formula G is called an *interpretation* of G.

An interpretation I for a formula G can represented as the following list of literals:

$$\{ L | I \models L, L \text{ or } \neg L \text{ appears in } G \}$$

In this dissertation, we represent an interpretation as such a list.

Definition 7. A formula G is *true under an interpretation* I if and only if G is evaluated to **T** in the interpretation. Then we say that I is a *model* of G and I *satisfies* G; otherwise, G is *false under* I and I *falsifies* G.

Example 1. $G = (p \vee q)$ is true under interpretations $\{ p, q \}$, $\{ p, \neg q \}$, and $\{ \neg p, q \}$. But G is not true under interpretation $\{ \neg p, \neg q \}$. $F = (p \vee \neg p)$ is true under all interpretations ($\{ p \}$ and $\{ \neg p \}$). □

We now give some important definitions regarding the truth value of a formula in various interpretations.

Definition 8. A formula is *valid* (a *tautology*) if and only if it is true under all its interpretations. A formula is *unsatisfiable* if and only if it is false under all its interpretations. A formula is *satisfiable* if and only if it is true under at least one interpretation (having at least one model).

Example 2. In the previous example, F is valid and G is satisfiable. $H = (p \wedge \neg p)$ is unsatisfiable since it is false under either $\{ p \}$ or $\{ \neg p \}$. □

Theorem proving in general is to decide if a formula is unsatisfiable (or valid). This can be achieved by many procedures which will be examined in detail in later sections.

2.2 First-Order Logic

We now give a brief review of first-order (predicate) logic which our methods are based on. First-order logic is more general than the propositional logic and has more expressive power. Again, in the review, we present syntax and semantics of first-order logic and related topics.

2.2.1 Syntax

First-order logic has more expressive power than the propositional calculus in that it allows notions of a countable set of individual objects and related mappings and relationships on the objects.

We begin with some examples. Suppose we want to say John is the father of Mary; in first-order logic it can be represented as $father(John, Mary)$ which specifies the relationship that John is the father of Mary. Similarly $greater(1, 2)$ can be used to represent $1 > 2$.

Often we also would like to specify mappings among objects. For example, we can use $plus(1, 2)$ to represent the value of $1 + 2$ (by mapping the pair $(1, 2)$ to value of $1 + 2$); similarly $wife(John)$ represents John's wife. Mappings like $plus$ represent some object corresponding to some given ones.

Definition 9. Individual objects, such as "John" and "1", are *constants*. The mappings, such as "plus" and "wife," are *functions*. A function specifies a mapping from a list of objects to an object. The properties, such as "father" and "greater," are *predicates*. A predicate specifies a relationship among objects. Functions and predicate have *arguments* on which the mappings or relationships are specified.

To make general statements like "every dog eats something" which apply to all dogs and something that each individual dog eats, we need a mechanism to represent more than one individual object.

Definition 10. A *variable* represents more than one object and is characterized by one of two *quantifiers* ($\forall$ and $\exists$) as described in Table 2.4.

$\forall$	for all (*universal quantifier*)
$\exists$	there exists (*existential quantifier*)

Table 2.4: Quantifiers in first-order logic

For example, we might use $\forall x \exists y (dog(x) \supset eats(x, y))$ to represent "every dog eats something."

A predicate can have variables, constants and functions as its arguments.

Definition 11. *Terms* are defined as follows:

(1) A constant is a term;

(2) A variable is a term;

(3) If f is an n-place function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term;

(4) All terms are generated by applying the above rules. □

As in the propositional calculus, we define atoms as follows:

Definition 12. If p is an n-place predicate symbol, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom*.

Definition 13. A *literal* is an atom or the negation of an atom; an atom is a *positive literal* and the negation of an atom is a *negative literal*.

The logical connectives described in Table 2.1 are also used with atoms and quantifiers in first-order logic to construct (well-formed) formulas.

Definition 14. Well-formed formulas in first-order logic are defined as follows:

(1) An atom is a formula;

(2) If F is a formula, then $\neg F$ is a formula;

(3) If F and G are formulas, then $F \vee G$, $F \wedge G$, $F \supset G$, and $F \equiv G$ are formulas;

(4) If F is a formula and x is a variable, then $(\forall x)F$ and $(\exists x)F$ are formulas;

(5) All formulas are generated by applying the above rules. □

For example, $\forall x \forall y \exists z (p(x, y) \supset q(z))$ and $\exists x \forall y (p(x, y) \wedge r(x, f(x)))$ are formulas, in which p, q and r are predicates and x, y, z and $f(x)$ are terms.

In particular, we are interested in formulas having no variables.

Definition 15. A formula (or a term) like $p(a,b) \wedge q(a)$ with *no* variables is *ground.* For example, formula $p(a,b) \vee \neg q(a) \vee \neg q(b)$ is ground and contains one positive and two negative literals.

Ground atoms can be seen as atoms in the propositional calculus. Indeed, first-order logic can be considered as an extension of the propositional logic, so ground formulas (containing no variables) in first-order logic can be treated as formulas in the propositional logic. This concept is fundamental to understand our methods.

Following convention in the literature, we use u, v, w, x,... for variable names (sometimes in upper-case to distinguish from constants); a, b, ... for constant names (always in lower-case); $f(\ldots)$, $g(\ldots)$, ... for function symbols; and $p(\ldots)$, $q(\ldots)$, ... for predicate symbols.

2.2.2 Semantics

As in the propositional calculus, semantics assigns meanings to all the symbols in a formula. However, since first-order logic allows variables which range over more than one object, when interpreting a formula, we must specify the *domain* as well as the *interpretation* of constants, functions and predicates with respect to that domain. Note that in the propositional calculus, only interpretations are needed.

Definition 16. A *structure* $I = (\mathcal{D}, \mathcal{I})$ of a formula F in first-order logic consists of a nonempty domain $\mathcal{D}$, and an interpretation $\mathcal{I}$ (assignment of "values") for each constant, function and predicate occurring in F as follows:

(1) Each constant is assigned an element in $\mathcal{D}$;

(2) Each n-place function is assigned a mapping from $\mathcal{D}^n$ to $\mathcal{D}$, in which $\mathcal{D}^n = \{(x_1, \ldots, x_n) | x_1 \in \mathcal{D}, \ldots, x_n \in \mathcal{D}\}$;

(3) Each n-place predicate is assigned a mapping from $\mathcal{D}^n$ to $\{\mathbf{T}, \mathbf{F}\}$ (the *truth value* of this predicate). □

The evaluation of formulas for a structure I is defined as follows.

Definition 17. With a structure $I = (\mathcal{D}, \mathcal{I})$, a formula can be *evaluated* to $\mathbf{T}$ or $\mathbf{F}$ by the following rules:

(1) If the truth values of formulas G and H are evaluated, then the truth values of the formulas $\neg G$, $(G \wedge H)$, $(G \vee H)$, $(G \supset H)$, and $(G \equiv H)$ are evaluated according to Table 2.3.

(2) $(\forall x)G$ is evaluated to **T** if G is evaluated to **T** for every x in $\mathcal{D}$; otherwise, it is evaluated to **F**.

(3) $(\exists x)G$ is evaluated to **T** if G is evaluated to **T** for at least one x in $\mathcal{D}$; otherwise, it is evaluated to **F**. □

For example, consider the formula $\forall x(p(x, s(x)))$. We can have a structure I which has an integer domain, assigns normal meanings to numerical constants, and interprets function $s(x)$ as $x+1$ and predicate $p(x, y)$ as $x < y$. With this structure, this formula has a truth value **T**. For simplicity, we always represent a first-order structure I by a (probably infinite) set of *all* ground literals that are true in I. In particular, we are interested in *Herbrand structures* that can be constructed from the given formula in clause form (see Section 2.4.2 for more details).

We can further distinguish different structures by their domains.

Definition 18. A structure is *infinite* if it has an infinite domain; *finite* if it has a finite domain.

For example, a structure of the group Z_4 will be finite; the one on real intervals is infinite.

As in the propositional calculus, we have the following definitions regarding formula truth values in different structures.

Definition 19. A formula F is *satisfiable* if there exists some structure I that evaluates F to **T**; we then say I *satisfies* F ($I \models F$) and I is a *model* of F. If no structure can be found to make F true, we say F is *unsatisfiable*. F is *valid* if F is true under *any* structure.

For example, the formula $\forall x(p(x, s(x)))$ is satisfiable because we have seen a structure to make it true. But it is not valid because it is not true when $s(x)$ is interpreted as $x - 1$ and $p(x, y)$ as $x < y$. Formula $\forall x(q(x) \wedge \neg q(x))$ is (obviously) unsatisfiable; formula $\forall x(q(x) \vee \neg q(x))$ is valid.

Definition 20. A formula F is a *logical consequence* of (or *logically follows from*) another formula G if F is true in all structures that make G true; we write $G \models F$.

In other words, $G \models F$ means that F is true whenever G is true. Thus it is not difficult to see that $G \wedge \neg F$ is unsatisfiable. For example, $p(a) \models \exists x(p(x))$ is true and $p(a) \wedge \neg \exists x(p(x))$ is clearly unsatisfiable.

Usually there are many different "forms" for a formula, we are particularly interested in the equivalence among those formulas.

Definition 21. Two formulas F and G are *equivalent*, denoted by $F = G$, if and only if the truth values of F and G are the same under every structure.

Informally, we can think of a model as an *example* as used in [BB82, Pla84, Wan85]. Use of examples in this way is similar to the way mathematicians use diagrams (or examples) to help solve hard problems. However, we use semantics in a more general way and this will be discussed in Chapter 6.

2.2.3 Clause Form

In this section, we discuss a very important standard form, the *clause form*, for automated first-order logic theorem proving (to be discussed in detail in Section 2.4). Every first-order logic formula F can be transformed to clause form which preserves the satisfiability of F. However, unlike normal forms, the formula in clause form is not equivalent to the original formula.

As in the propositional calculus, a formula in first-order logic can be transformed to an equivalent normal form. In addition to the conjunctive normal form (CNF) and disjunctive normal form (DNF), there is another normal form in first-order logic regarding the placement of quantifiers.

Definition 22. A formula F in first-order logic is in *prenex normal form* if and only if the formula F is in the form of

$$(Q_1 x_1) \cdots (Q_n x_n)(M)$$

where every $(Q_i x_i)$, $i = 1, \ldots, n$, is either $(\forall x_i)$ or $(\exists x_i)$, and M is a formula containing no quantifiers.

All the previous examples are in prenex normal form; $\forall x(p(x) \supset \exists y(r(x,y)))$ is not. Every first-order formula can be converted into an equivalent formula in prenex normal form. Prenex normal form is used to simplify proof procedures. We will not discuss how to transform a formula into a prenex normal form; such methods can be found in [CL73, Lov78].

Given a first-order formula F in prenex normal form, we use a procedure called *Skolemization* to replace all the existentially quantified variables by *Skolem functions*, and obtain a quantifier-free first-order formula F' (in which all variables are implicitly universally quantified). F' is said to be in *Skolem standard form*.

Skolem functions express the dependency of existentially quantified variables on those universally quantified variables appearing before them. If there are no such universally quantified variables, *Skolem constants* are introduced. We will not discuss the Skolemization procedure, which is described in detail in [CL73, Lov78].

Example 3. In the following formula

$$(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)p(x,y,z,u,v,w)$$

we replace x by a Skolem constant a, u by a Skolem function $f(y,z)$ and w by a Skolem function $g(y,z,v)$. And the following formula in Skolem standard form is obtained

$$(\forall y)(\forall z)(\forall v)p(a,y,z,f(y,z),v,g(y,z,v))$$

Note that functions f and g respectively express the dependency of u on y and z, and the dependency of w on y, z, and v. Variable x does not depend on any others and can be replaced by a new constant a. □

For a formula F in Skolem standard form, we can transform F into a conjunctive normal form (CNF) as we do in the propositional calculus. Note that in such CNF, all variables are universally quantified so quantifiers can be omitted. Also, each disjunction constitutes a special "unit," and the CNF can be seen as a conjunction of such special units. With Definition 13 on literals, we can define such units as follows:

Definition 23. A *clause* is a disjunction of literals. A set S of clauses (or a *clause set*) is regarded as a conjunction of all clauses in S, where every variable in S is universally quantified. An empty clause $\{\,\}$ represents $\mathbf{F}$.

After transforming a formula F in Skolem standard form into CNF, we can think of it as containing several logical assertions put in conjunction. Such assertions can be, for example, mathematical axioms. Each assertion is a clause.

Since a clause contains a disjunction of literals, in later discussions when there is no confusion, we usually omit the disjunctions ($\vee$) and represent it as a set. Also, F is represented as a set of clauses (called a *clause set*) and the conjunctions ($\wedge$) are removed.

Example 4. For example, the formula

$$(p(a) \vee q(X)) \wedge (p(b) \vee r(Y)) \wedge (q(a) \vee r(Z))$$

can be represented by a clause set $\{\{p(a), q(X)\}, \{p(b), r(Y)\}, \{q(a), r(Z)\}\}$ (or $\{p(a) \vee q(X), p(b) \vee r(Y), q(a) \vee r(Z)\}$), which contains three clauses $\{p(a), q(X)\}$, $\{p(b), r(Y)\}$, and $\{q(a), r(Z)\}$. Variables X, Y, and Z are universally quantified and quantifiers are omitted. □

Since universal quantifiers distribute over $\wedge$, each clause in a clause set should have different variables and renaming is done when necessary.

Horn clauses are common and important in theorem proving.

Definition 24. A clause having *at most one positive literal* is called a *Horn clause.* A *Horn (clause) set* is a set of Horn clauses.

For example, $\{p(a,b), \neg q(a), \neg q(b)\}$ is a Horn clause. Many axiom systems can be expressed in Horn clauses which allows the use of many simpler proving techniques.

Definition 25. A *unit clause* is a clause with only one literal.

Unit clauses are of particular interest because they state single facts which are usually of immediate use.

Example 5. Consider the following first-order formula F:

$$\forall x((\forall y)p(x,y)) \supset \neg(\forall y)(q(x,y) \supset r(x,y))$$

We will illustrate how to translate F into a set of clauses.

Replace $\supset$ and $\equiv$ with $\vee$ and $\wedge$ This is based on the equivalent relations in Table 2.2.

$$\forall x \neg((\forall y)p(x,y)) \vee \neg(\forall y)(\neg q(x,y) \vee r(x,y))$$

Push in negations This step distributes negations so they apply only to atoms.

$$\forall x((\exists y)\neg p(x,y)) \vee \exists y(q(x,y) \wedge \neg r(x,y))$$

Convert to prenex normal form This step puts every quantifier in front of the formula.

$$\forall x \exists y \exists z(\neg p(x,y) \vee (q(x,z) \wedge \neg r(x,z)))$$

Convert to Skolem standard form Existentially quantified variables are replaced by Skolem functions and a Skolem standard form is obtained. In this case, y is replaced by $f1(x)$ and z is replaced by $f2(x)$.

$$\neg p(x, f1(x)) \vee (q(x, f2(x)) \wedge \neg r(x, f2(x)))$$

Convert to conjunctive normal form

$$[\neg p(x, f1(x)) \vee q(x, f2(x))] \wedge [\neg p(x, f1(x)) \vee \neg r(x, f2(x))]$$

Convert to clauses and rename variables We rename variables so no two conjuncts share variables. This is based on the following equivalence where $F[x]$ represents a formula F containing a variable x.

$$\forall x(F[x] \wedge H[x]) = (\forall x)F[x] \wedge (\forall x)H[x]$$

Thus this step generates two clauses and we are done.

$$\{\, \neg p(x1, f1(x1)), q(x1, f2(x1)) \,\}$$
$$\{\, \neg p(x2, f1(x2)), \neg r(x2, f2(x2)) \,\}$$

Such process is often referred to as *standard clause translation* and can be easily mechanized. However, such process is known to generate a huge number of clauses sometimes for simple first-order formulas. This is because the application of distributivity causes exponential explosion during the translation to conjunctive normal form. A non-standard structure-preserving clause translation has been proposed. The basic idea is to replace sub-formulas with new variables and assert at the top level related formulas stating the equivalence between new variables and corresponding sub-formulas [PG86]. □

Note that a first-order formula F and its corresponding clause form S are *not* logically equivalent. However, the usefulness of clause form in theorem proving lies in the fact that satisfiability of F is preserved by S, which is highlighted in the following theorem.

Theorem 26. *Let S be a set of clauses that is derived from a first-order formula F using the Skolemization process. Then F is unsatisfiable if and only if S is unsatisfiable.*

Proof. See [CL73] for a straightforward proof. □

Thus detecting unsatisfiability of a first-order formula is equivalent to detecting unsatisfiability of the corresponding formula in clause form. This is the basis of many clause form theorem proving techniques.

Note, however, that F is not equivalent to its clause form. Theorem 26 only concerns the preservation of satisfiability by clause form, which most proof procedures are about.

In later discussions, we will use first-order formulas only in the descriptions of the original theorems. Most of the time we will use clauses.

The logic of many commonly used theorem proving techniques (for example, resolution and hyper-linking) is a quantifier-free first-order logic in clause form. So a formula in first-order logic has to be converted into a set of clauses before being submitted to such theorem provers. Other proving techniques (for example, natural deduction [Fit90]) accept formulas in first-order logic directly.

2.2.4 Proof Procedures

Theorem proving in general shows that a formula T (the *theorem*) logically follows from another formula A (conjunction of the *axioms*), in which T and A are expressed properly in symbolic logic. A *proof* is a demonstration that T logically follows from A. If there exists a proof to derive T from A, we say $A \vdash T$. Note that this is *not* equal to saying $A \models T$ (to be discussed in Section 2.4.1). A procedure to find such proofs is a *proof procedure*. For example, given the group definition as the axioms, we can prove the theorem for the cancellation property. An *automated theorem prover* is a computer program that implements a proof procedure. In the following sections, we will introduce several proof procedures that are closely related to our proof procedures for first-order logic.

2.3 Propositional Calculus (PC) Prover

Theorem proving in the propositional calculus is best carried out by a *refutation* procedure which shows that $(A \wedge \neg T)$ is unsatisfiable when proving T logically follows from A (to be discussed in more detail in Section 2.4.1). The most obvious method is to evaluate the formula $(A \wedge \neg T)$ under all possible interpretations since there are only a finite number of interpretations for a formula (of finite length) in the propositional logic. There are more efficient decision procedures for the propositional calculus to determine, without examining all interpretations, whether a formula is unsatisfiable or not.

One well-known procedure is due to Davis and Putnam [DP60]. We now discuss a variant of the Davis-Putnam procedure by Lee and Plaisted [Lee90, LP92] since this variant is used in our theorem prover. For simplicity, we refer to this variant as the *propositional calculus (PC) prover.*

The PC prover takes a set of ground clauses as input and applies (recursively) case analysis and simplification rules. The basic idea is [1] described in Fig. 2.1.

There are four simplification rules, described below, that are applied to each

[1]PC prover is actually more complicated and uses some methods, for example the *cut-off* strategy and an early satisfiability check, to intelligently prune the semantic tree. We only present the simple concept here.

```
Algorithm PC_Prover(S)
  input
    S: a set of ground clauses
  output
    None
  return Values
    T: S is satisfiable
    F: S is unsatisfiable
  begin
    if S is F or T then
      return S
    else
      % Start case analysis
      Pick one literal L from S
      % First case:  T
      Assume L to be T and simplify S to S'
      if PC_Prover(S') is T then
        return T
      % PC_Prover(S') is F
      else
        % Second case:  F
        Assume L to be F and simplify S to S'
        return PC_Prover(S')
  end
```

Figure 2.1: Propositional calculus prover algorithm

clause. S is a conjunction of clauses and A is a disjunction of literals. *Unit deletion* says that a clause containing **T** may be deleted; *unit simplification* says that **F** in any clause may be deleted. The other two are obvious.

$$\frac{S \wedge (A \vee \mathbf{T})}{S}(\text{Unit Deletion}) \tag{2.1}$$

$$\frac{S \wedge (A \vee \mathbf{F})}{S \wedge A}(\text{Unit Simplification}) \tag{2.2}$$

$$\frac{S \wedge \mathbf{T}}{S} \tag{2.3}$$

$$\frac{S \wedge \mathbf{F}}{\mathbf{F}} \tag{2.4}$$

The PC prover process is best explained as a preorder traversal of a *semantic tree*.

Definition 27. Given a set S of ground clauses, let A be the atom set of S. A *semantic tree* for S is a binary tree T, where

(1) each edge is labeled with an atom or negation of an atom from A,

(2) each non-terminal node contains two subtrees linked by two edges labeled with L and $\neg L$ respectively, where $L \in A$,

(3) each node represents a clause set (with S for the root) which is obtained by applying simplification rules on the clause set at the parent node using the literal at the edge linking this node and the parent node,

(4) and any path from the root contains neither complementary pair nor duplicate pair.

A maximal path from the root is called a *branch*. Note that a branch might be infinite (not terminating) if S is not finite. □

Each node of a semantic tree can be seen as containing two cases (as shown in Figure 2.1): **T** and **F** of a literal L. At each node, PC prover simplifies the clause set using the simplification rules based on the case (truth value) of L leading to the node. The semantic tree is not unique due to the different orderings to choose literals for case analysis. The rule used to determine which literal to choose for case analysis, is called the *branching rule*.

There are several important properties of a semantic tree regarding a proof search by a PC prover on a clause set.

Definition 28. A path of a semantic tree *contradicts* a clause C if, for each literal $L \in C$, $\neg L$ appears in the path. C is a *contradicted clause*. A node N is a *failure node* or a *contradiction node* if the path from the root to N contradicts some clause.

Definition 29. A branch B of a semantic tree is *closed* if and only if B terminates at a failure node. A semantic tree T is *closed* if and only if every branch of T is closed. In this case, the original clause set S is unsatisfiable.

Obviously, all children nodes of a failure node are also failure nodes. In this case, we usually mark the one which has no failure nodes as ancestors. Sometimes, we mark failure nodes corresponding to the nodes containing **F**.

A semantic tree has a direct relation with the satisfiability of the original clause set S to which the PC prover is applied.

Remark. If a branch of a semantic tree terminates at a node containing **T**, all the literals on the branch constitute a model for the clause set S (which is satisfiable). If the semantic tree is closed, S is unsatisfiable.

Thus a PC prover can be easily modified to a *model searching* procedure: A model is returned for the input clause set S if S is found satisfiable. This is an option in Lee's prover. However, our prover basically does model searching and makes use of the returned models to further guide the proof search. We will discuss this in Chapter 4.

Example 6. For the following clause set S

$C1: \; \{\neg p, q\}$
$C2: \; \{\neg p, \neg q\}$
$C3: \; \{p, q\}$
$C4: \; \{a, \neg b\}$
$C5: \; \{\neg a, \neg b\}$
$C6: \; \{a, b\}$
$C7: \; \{\neg a, b\}$

One possible semantic tree is presented in Fig. 2.2. At each node the set of simplified clauses is shown, which is obtained by applying the simplification rules to the parent node based on the chosen case of a literal. For example, by assuming p to be **F**, the original clause set S is simplified to $\{C3', C4, C5, C6, C7\}$ (as shown by the left son of the root).

In the semantic tree, "$\times$" indicates a failure node; the input clauses contradicted by branches are shown after the "$\times$." For example, the leftmost "$\times$" indicates that $C5 = \{\neg a, \neg b\}$ is contradicted by the branch $\{\neg p, a, b\}$ along the leftmost path.

The semantic tree is closed and shows that S is unsatisfiable; the PC prover will return **F** with S as input. $\square$

2.4 First-Order Theorem Proving

Since there are an infinite number of structures in first-order logic, evaluating a formula under all structures to determine validity or satisfiability, will be impossible. Thus proof procedures are needed to carry out such tasks. In this section we give an

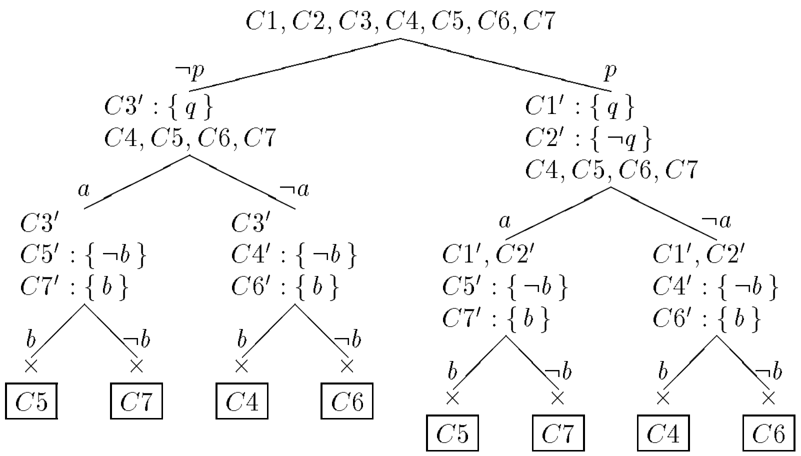


Figure 2.2: A semantic tree for Example 6

brief review of first-order logic theorem proving. Some basic materials are discussed first, followed by the discussions of two important theorem proving strategies related to our prover.

2.4.1 The Basics

A *first-order logic automatic theorem prover* is a computer program that can prove that, in first-order logic, a theorem logically follows from other axioms. Unlike the propositional calculus, first-order logic is only *semi-decidable* [WOLB84, WR73] in that any first-order logic theorem proving procedure will find a proof if a theorem is given, but it might not terminate if a non-theorem is given.

In previous sections we mentioned that some theorem proving procedures take clause sets as input. Those are *clause form* theorem proving procedures. Although the clause set is not logically equivalent to the original formula, according to Theorem 26, the clause set is unsatisfiable if and only if the original first order formula is unsatisfiable. So it is sufficient just to consider the clause set in search for a proof of unsatisfiability. There are also non-clause form proving procedures (such as *semantic tableaux* [Fit90]) which will not be covered. In the following discussion, we only discuss clause form theorem proving.

First, we present the mechanism a theorem prover uses to do *inference*. Here is a general definition.

Definition 30. A *inference rule* contains a set of sentence patterns called *conditions*, and another set of sentence patterns called *conclusions*. It is usually presented as

$$\frac{conditions}{conclusions}$$

Whenever sentences *match* the conditions of the inference rule, it is acceptable to infer sentences matching the conclusions. Such process of using inference rules is called *inference*. □

Inference rules are used in the proofs to derive new clauses that logically follow from the existing ones. For example, the following is the *modus ponens* inference rule commonly used by the mathematicians to prove mathematical theorems:

$$\frac{F \quad F \supset G}{G} \tag{2.5}$$

To apply inference rules, we have to do "matching" among the clauses. In the propositional calculus, this is straightforward; the simplification rules in Section 2.3 are actually inference rules in the PC prover. However, in first-order logic, it is more complicated. The reason is that a variable can represent any term in the corresponding domain. This is formalized as follows.

Definition 31. A *substitution* is a finite set of the form $\{ v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n \}$ where every v_i is a variable and every t_i is a term to replace the corresponding v_i. When $t_1, \ldots, t_n$ are all ground terms, the substitution is called a *ground substitution*. A substitution containing no elements is called the *empty substitution*. Substitutions are usually represented by Greek letters such as θ, α, and β.

For example, $\{ x \leftarrow f(a) \}$ is a ground substitution and substitutes x by function $f(a)$; $\{ x \leftarrow y, z \leftarrow g(y) \}$ replaces x by another variable y, and variable z by function $g(y)$.

Furthermore, new substitution can be obtained by composing two substitutions.

Definition 32. The *composition* of substitutions θ and σ, denoted by $\theta \circ \sigma$, is obtained by first applying σ to the *term* for all $x \leftarrow term$ of θ, and then taking the union of substituted θ and $\sigma - \{ x \leftarrow term | x \leftarrow term \in \sigma$ and there exists some $x \leftarrow term1$ in $\theta \}$. Composition of substitutions can also be thought of as applying substitutions in sequence.

For example, for $\theta_1 = \{ x \leftarrow g(y) \}$ and $\sigma_1 = \{ y \leftarrow f(a) \}$, $\theta_1 \circ \sigma_1 = \{ x \leftarrow g(f(a)), y \leftarrow f(a) \}$; for $\theta_2 = \{ x \leftarrow a \}$ and $\sigma_2 = \{ x \leftarrow b \}$, $\theta_2 \circ \sigma_2 = \{ x \leftarrow a \}$.

Definition 33. Applying a substitution θ to a clause C generates an *instance* $C\theta$ of C. If the instance is ground, we say that a *ground instance* is obtained.

For example, if we apply $\{ x \leftarrow a, y \leftarrow c \}$ to $\{ p(x), \neg q(b, x, y) \}$, we obtain a ground instance $\{ p(a), \neg q(b, a, c) \}$.

Instances can be seen as "instantiations" of variables by substitutions. Thus in first-order logic we can substitute a variable by another term, namely we can apply a substitution to a variable, in matching the conditions of the inference rules.

Example 7. Suppose we have the clause $\{ \neg dog(x), barks(x) \}$ (from $\forall x(dog(x) \supset barks(x))$) to represent "every dog barks," and $\{ dog(dopey) \}$ to represent "dopey is a dog." Using the modus ponens inference rule and replacing variable x in the first formula with *dopey* we can infer that $barks(dopey)$ ("dopey barks").

In the above example, $dog(x)$ and $dog(dopey)$ are made equal by the substitution $\{ x \leftarrow dopey \}$. This process is quite often used in applying inference rules (especially matching the conditions of the inference rules).

Definition 34. The process of making two or more literals $L_1, \ldots, L_n$ equal by applying the same substitution to each one, is called *unification* and the corresponding substitution is called a *unifier*. Literals $L_1, \ldots, L_n$ are said to be *unifiable*.

So in Example 7, $dog(x)$ and $dog(dopey)$ are unifiable and can be unified by the unifier $\{ x \leftarrow dopey \}$.

Usually there might be more than one unifier to unify literals. For example, substitutions $\theta = \{ x \leftarrow g(z) \}$ and $\sigma = \{ x \leftarrow g(a) \}$ both make literals $p(x)$ and $p(g(y))$ equal up to variable renaming. In this case, we are interested in the most general one θ.

Definition 35. A unifier θ for literals $L_1, \ldots, L_n$ is a *most general unifier* (*mgu*) if and only if for each unifier α for those literals there is a substitution β such that $\alpha = \theta \circ \beta$.

So θ is the mgu in the above example and $\sigma = \theta \circ \{ y \leftarrow a \}$. A mgu can be seen as the "least" substitution that should be carried out to unify literals. A unification algorithm to find a most general unifier is given in [CL73, Lov78].

Now we know in general how to apply inference rules, especially through unification to create identity, for example, pairs of equal literals. There are several important issues in automated first-order logic theorem proving.

A theorem prover uses either an *affirmation* or a *refutation* procedure to prove that a theorem T logically follows from axioms A. An affirmation procedure shows that $A \supset T$ is valid; a refutation procedure shows that the negation of the theorem together with the axioms, $A \wedge \neg T$ constitute a contradiction. A refutation procedure can be thought of as the "proof by contradiction" technique often used by human to prove mathematical theorems. Since clauses retain the satisfiability of the original formula, they are ideal candidates for refutation procedure, namely to prove that the clause form of $A \wedge \neg T$ is unsatisfiable. Our prover uses such a refutation procedure. So we will only discuss and focus on related topics on proofs by refutation.

The direction of applying inference rules is another important issue. A theorem prover can work in *backward chaining* or *forward chaining* style. Forward chaining reasons from the axioms to prove the theorem is valid (in affirmation procedures) or the negation of the theorem is contradicted (in refutation procedures); backward chaining starts with the theorem or the negation of the theorem as a *goal* and generate *subgoals* by applying inference rules backward until subgoals match axioms. For example, in the modus ponens inference rule (2.5), forward chaining generates G from F and $F \supset G$; backward chaining generates two subgoals F and $F \supset G$ from the goal G.

There are two very important criteria about theorem proving procedures. In general inference rule application is purely a *syntactic* operation. Recall that in Section 2.2.4 we introduced the notion $A \vdash T$ if there exists a proof to derive T from A. Such a syntactic derivation intends to show that *semantically* $A \models T$. The relation between the syntactic derivation and the semantic entailment determines the "quality" of a theorem proving technique in two aspects: One is *completeness*, namely a theorem prover should always find a proof whenever a theorem logically follows from the axioms. Formally completeness guarantees that

$$A \models T \supset A \vdash T$$

The other is *soundness*, namely a theorem prover should only find a proof whenever the theorem logically follows from the axioms. Formally soundness guarantees that

$$A \vdash T \supset A \models T$$

Soundness must be assured and fortunately is easy to achieve, but completeness is not. Some proving procedures are not complete but are still useful (mostly because they are complete for a subclass of first-order logic which is, though limited, often useful); we will see some examples of such techniques later.

In the following two sections, we will briefly introduce two clause form refutation theorem proving procedures—the *resolution* and *instance-based* procedures—that are closely related to our methods.

2.4.2 Theoretical Foundation

One major theoretical base for clause form refutation theorem proving is Herbrand's Theorem [Her74, CL73, Pla90].

Theorem 36 (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. See for example [CL73]. □

Based on Herbrand's Theorem, the unsatisfiability test of a clause set S in first-order logic can be reduced to the test of the ground instance set S' (in the propositional calculus). How can the ground instances be obtained? In other words, what is the domain of S? There could be countless of them but it is sufficient just to examine one domain for testing satisfiability: the *Herbrand universe*.

Definition 37 ([Lov78]). Let S be a set of clauses. The *Herbrand universe* H of S is defined inductively as follows:

(1) any constant symbol of any formula of S is a member of H (if no constant occurs in S, then H contains a single constant, say a);

(2) if f is an n-place function symbol of S and terms $t_1, \ldots, t_n$ are in H, then $f(t_1, \ldots, t_n)$ is in H.

The elements in the Herbrand universe are *ground terms*. □

Example 8. For example, the Herbrand universe for clause set $\{ p(X) \vee q(X), r(f(Y)) \}$ is

$$\{ a, f(a), f(f(a)), f(f(f(a))), \ldots \},$$

and the Herbrand universe for clause set $\{ p(f(X)), q(g(b, X)) \}$ is

$$\{ b, f(b), g(b, b), f(f(b)), g(b, f(b)), g(f(b), b), g(f(b), f(b)), \ldots \}$$

□

The Herbrand universe provides a domain from which ground atoms can be constructed for a clause set. The idea is to replace variables by the terms in the Herbrand universe.

Definition 38. Let S be a set of clauses. The set of ground atoms of the form $p(t_1, \ldots, t_n)$, for all n-place predicates $p(s_1, \ldots, s_n)$ occurring in S, obtained by replacing all variables in $p(s_1, \ldots, s_n)$ with elements of the Herbrand universe of S, is called the *atom set*, or the *Herbrand base* of S. □

Thus the Herbrand base can be used to generate the ground instances of the clauses of a given clause set S and apply Herbrand's Theorem to test the satisfiability of S.

Definition 39. An *enumeration* of the Herbrand base is a way to obtain all atoms by uniformly replacing each variable of an atom of a literal occurring in S by a member of the Herbrand universe.

Example 9. Continuing from Example 8, the Herbrand base for $\{ p(X) \vee q(X), r(f(Y)) \}$ is

$$\{ p(\underline{a}), q(\underline{a}), p(\underline{f(a)}), q(\underline{f(a)}), r(f(\underline{a})), p(\underline{f(f(a))}), q(\underline{f(f(a))}), r(f(\underline{f(a)})), \ldots \}$$

and the Herbrand base for $\{ p(f(X)), q(g(b, X)) \}$ is

$$p(f(\underline{b})), q(g(b, \underline{b})), p(f(\underline{f(b)})), q(g(b, \underline{f(b)})), p(f(\underline{f(f(b))})), q(g(b, \underline{f(f(b))})), \ldots$$

respectively. Underlined terms are those from the Herbrand universe to replace variables. □

Through the enumeration of the Herbrand base, one can systematically generate ground instances and obtain a straightforward method for clause form refutation theorem proving based on Herbrand's Theorem. There are several ways to enumerate the Herbrand base. One is to enumerate in the order of increasing length, when written as a string (for example, "$pfab$" for $p(f(a, b))$ has length 4), of the atoms. In Example 9, such enumeration is used. In later discussions, we always assume such Herbrand base enumeration is used. Also, we refer to ground literals (or clauses) as those constructed from the Herbrand base.

We can also modify the definition for structures of a clause set with respect to the Herbrand universe.

Definition 40. A *Herbrand structure* I for a clause set S is a structure $(\mathcal{D}, \mathcal{I})$ such that:

(1) $\mathcal{D}$ is the Herbrand universe H for S,

(2) $\mathcal{I}$ maps each constant into itself

(3) Let f be an n-place function and $h_1, \ldots, h_n$ be elements of H. In $\mathcal{I}$, f is assigned a mapping from $(h_1, \ldots, h_n)$ ($\in H^n$) to $f(h_1, \ldots, h_n)$ ($\in H$).

(4) In $\mathcal{I}$, each n-place predicate is still assigned a mapping from H^n to $\{ \mathbf{T}, \mathbf{F} \}$.

A Herbrand structure $I = (\mathcal{D}, \mathcal{I})$ can be conveniently represented by a set (probably infinite)

$$\{ L_1, L_2, \ldots, L_n, \ldots \}$$

in which each L_i is either A_i or $\neg A_i$ for every $A_i \in \mathcal{D}$ such that if L_i is A_i, A_i is true in I; otherwise A_i is false in I[2]. We will represent a Herbrand structure as such a set whenever possible, and we will simply call it a structure if there is no confusion.

Evaluation of clauses based on a Herbrand structure I is defined as follows[3].

Definition 41. Given a Herbrand structure I for a clause set S, the clauses in S can be evaluated by the following rules:

(1) An atom (positive literal) A is true in I ($I \models A$) if and only if A is evaluated to $\mathbf{T}$ by $\mathcal{I}$; otherwise, A is false in I ($I \not\models A$).

(2) A negative literal $\neg A$ is true in I if and only if A is false in I. This is denoted as $I \models \neg A$; otherwise, $\neg A$ is false in I.

(3) A ground clause $L_1 \vee \ldots \vee L_m$ is true in I if and only if at least one L_i is true in I; otherwise, it is false in I.

(4) A clause C is true in I ($I \models C$) if and only if *every* ground instance $C\sigma$ of C is true in I; otherwise, C is false in I ($I \not\models C$).

(5) A set S of clauses is true in I ($I \models S$) if and only if each clause in S is true in I; otherwise, S is false in I ($I \not\models S$).

Definition 42. If a clause set S is true in a Herbrand structure I, then I is a *Herbrand model* of S.

One important property of structures is *consistency* which ensures each atom A and $\neg A$ do not have the same truth value.

Definition 43. A structure $I = (\mathcal{D}, \mathcal{I})$ is *inconsistent* if there is an atom A in $\mathcal{D}$ such that both A and $\neg A$ are true in I. I is *consistent* if it is not inconsistent.

Note that a Herbrand structure is always consistent according to Definition 41. However, a user-specified structure is needed in our prover and such a structure might be inconsistent. Since inconsistent structures might make our proving methods fail

[2] Some represent a Herbrand structure by a subset of the Herbrand base, namely, a set of all the atoms in the Herbrand base that are true in I; those atoms not in the set are false in I

[3] It is easier to illustrate the evaluation if Herbrand structures are represented as sets of all atoms in the Herbrand base that are evaluated true [AvE82].

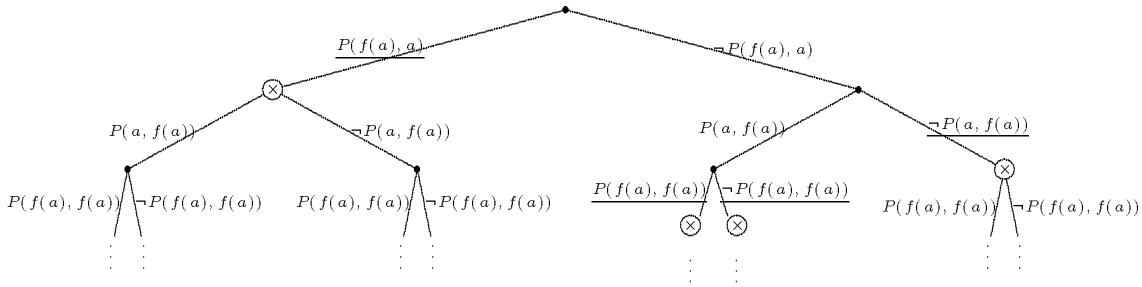


Figure 2.3: A semantic tree for the clause set $\{\neg p(f(Z), f(W)),\ p(f(a), a), p(X, f(Y))\}$.

to find a proof if there is one, in this thesis, we always assume a structure to be consistent if not otherwise specified.

With the notion of the Herbrand base, one can define (with Definition 27) the semantic tree of first-order logic. Note that now the atom set is the Herbrand base, and the labeling of the edge can be based on the Herbrand base enumeration. In a semantic tree for a clause set S, each branch corresponds to a Herbrand structure $\{ L_1, L_2, \ldots, L_n, \ldots \}$ for S with each L_i appearing on the branch. Failure nodes (or contradiction node), a path contradicting a clause (Definition 28), and a closed semantic tree (Definition 29) are defined similarly as in the propositional calculus.

Example 10. For the clause set $S = \{ \{ \neg p(f(Z), f(W)) \}, \{ p(f(a), a) \}, \{ p(X, f(Y)) \} \}$, a semantic tree can be constructed as shown in Fig. 2.3. If each left edge is attached a positive literal, the leftmost branch then corresponds to the Herbrand structure

$$\{ p(f(a), a), p(a, f(a)), p(f(a), f(a)), p(a, f(f(a))), \ldots \}$$

In particular, this semantic tree has four failure nodes (denoted by circled $\times$) and all branches are closed—the semantic tree is closed and S is unsatisfiable. The four corresponding paths contradict clauses in S by the underlined literals—from left to right, the contradicted clauses are $\{ \neg p(f(a), a) \}$, $\{ \neg p(f(Z), f(W)) \}$, $\{ p(X, f(Y)) \}$, and $\{ p(X, f(Y)) \}$ respectively. □

There is another important theorem regarding satisfiable clause sets in both the propositional and first-order cases.

Theorem 44 (Compactness Theorem). *A set S of formulas is satisfiable if and only if every finite subset T of S is satisfiable.*

Proof. See for example [Lov78]. □

In particular, we can apply the Compactness Theorem to ground clause sets. For example, in Lee's PC prover [Lee90], the *cut-off* idea based on the dependency sets is related to the above theorem. The cut-off idea also corresponds to the *pure literal rule* in Davis-Putnam procedure. In later sections, we will use the Compactness Theorem in discussions about satisfiability of ground clause sets.

2.4.3 The Resolution Procedure

Resolution [Rob65b] is an inference rule which takes two clauses and produces a third clause that logically follows from them. The process is to identify and remove complementary pairs of two clauses—one from each clause—using unification, and combine the other literals together to form a new clause.

The basic resolution procedure, *binary resolution*, can be described by the following inference rule:

$$\frac{C1 \quad C2}{(C1\theta - L\theta) \cup (C2\theta - M\theta)} L \in C1, M \in C2, L\theta = \neg M\theta \ with \ mgu \ \theta \tag{2.6}$$

$C1$ and $C2$ are the *parent clauses*; L and M are the *literals resolved upon.* $(C1\theta - L\theta) \cup (C2\theta - M\theta)$ is called the *resolvent.* To simplify discussions, we omit *factoring* [CL73, Lov78] which is needed to make resolution complete.

If clauses are seen as assertions, one way to view resolution is that a "contradiction" (complementary pairs after unification, for example, $L\theta$ and $M\theta$) is resolved and deleted from two formulas ($C1\theta$ and $C2\theta$) to generate a resolvent which can be seen as a new assertion. Resolution can also be seen as an extension of the modus ponens inference rule.

In a resolution theorem prover, resolution is applied to derive a sequence of clauses and derived clauses can be used in later resolutions. Resolution continues until an empty clause (**F**) is generated. A derivation of falsehood (an empty resolvent) from a clause set S is a *refutation*, or a *proof* for S. So resolution is a refutation procedure to derive unsatisfiability, and negation of the theorem has to be added to the axioms (in clause form) in search for a proof.

From the viewpoint of Herbrand's Theorem, resolution can be seen as applying in parallel the instances generation (by unification) and unsatisfiability check (by resolving upon contradicted literals).

Example 11. The following is a resolution proof with input clauses $C1$–$C4$. Clauses $C5$–$C7$ are derived resolvents.

$C1 : \{\neg s(Y), c(Y)\}$

$C2 : \{s(b)\}$

$C3 : \{v(a,b)\}$

$C4 : \{\neg c(Z), \neg v(a,Z)\}$

$C5 : \{c(b)\}$ from $C1$ and $C2$, $\theta = \{Y \leftarrow b\}$

$C6 : \{\neg v(a,b)\}$ from $C5$ and $C4$, $\theta = \{Z \leftarrow b\}$

$C7 : \{\}$ from $C6$ and $C3$

One can also look at this resolution proof from the viewpoint of Herbrand's Theorem: When generating $C5$, the resolution procedure generates a ground instance $\{\}$ of $C1$ and carries out one step of the satisfiability check by resolving upon the complementary pair $(s(b), \neg s(b))$. When generating $C6$, a ground instance $\{\}$ of $C2$ is generated and another step of the satisfiability check is done by resolving upon $(c(b), \neg c(b))$. The final satisfiability check step found the proof from $C6$ and $C3$. $\Box$

Binary resolution is sound in that each resolvent is a logical consequence of corresponding parent clauses, and complete (with factoring) in that a proof can always be found if the input clause set is unsatisfiable. However, binary resolution easily derives too many resolvents and the search for proofs becomes inefficient. One major reason is that literals are combined from different clauses; non-Horn clause sets in particular suffer from this combinatorial explosion in the search space. There are many refinements of resolution for more efficient search for proof. Now we will briefly review some related to our research.

If we restrict that at least one of $C1$ and $C2$ in formula (2.6) should be unit clause, we actually apply the modus ponens rule. This restricted resolution is called *unit resolution*; it can be seen as unit deletion in first order logic. Unlike binary resolution, unit resolution is only complete for Horn sets [HW74]. Further refinement on unit resolution is *UR resolution* [WOLB84], which is a simultaneous or multi-step application of unit resolution to get a *unit* resolvent. The unit resolvents can be used in later UR resolution. UR resolution can avoid generating intermediate resolvents.

Another refinement of binary resolution is *P1-resolution* [Rob65a] or *positive resolution*. The restriction is that either $C1$ or $C2$ in formula (2.6) must be positive (containing only positive literals). *Negative resolution* can be defined similarly. P1-resolution or negative resolution is complete. Further refinement on P1-resolution is *hyperresolution* [Rob65a], which can be seen as a simultaneous or multi-step application of P1-resolution to get a *positive* resolvent. The positive resolvents can be used in later hyperresolution. Hyperresolution can avoid generating intermediate resolvents. Hyperresolution is also complete and is often used.

2.4.4 The Instance-Based Procedures

Herbrand's Theorem provides a straightforward approach for refutation theorem proving. Before we begin the discussion on this approach, let us first look at one example.

Example 12. Let S be

$C1: \quad \{\neg p(X), q(f(X), X)\}$

$C2: \quad \{p(g(b))\}$

$C3: \quad \{\neg q(Y, Z)\}$

It is easy to verify that S is unsatisfiable. One of the unsatisfiable sets of ground instances of clauses in S contains:

$C1': \quad \{\neg p(g(b)), q(f(g(b)), g(b))\} \quad (= C1\{X \leftarrow g(b)\})$

$C2': \quad \{p(g(b))\} \quad (= C2)$

$C3': \quad \{\neg q(f(g(b)), g(b))\} \quad (= C3\{Y \leftarrow f(g(b)), Z \leftarrow g(b)\})$

These instances can be obtained by replacing variables with terms in the Herbrand universe. If ground instances are generated from the Herbrand base in a systematic way, one unsatisfiable ground clause set will be generated according to Herbrand's Theorem. □

One obvious refutation theorem proving strategy is *instance-based* (or *Herbrand-based*) to enumerate the Herbrand base to generate ground instances of the input clauses and use a propositional logic prover to test the ground clause set for unsatisfiability (because ground literals in first-order logic can be treated as propositional

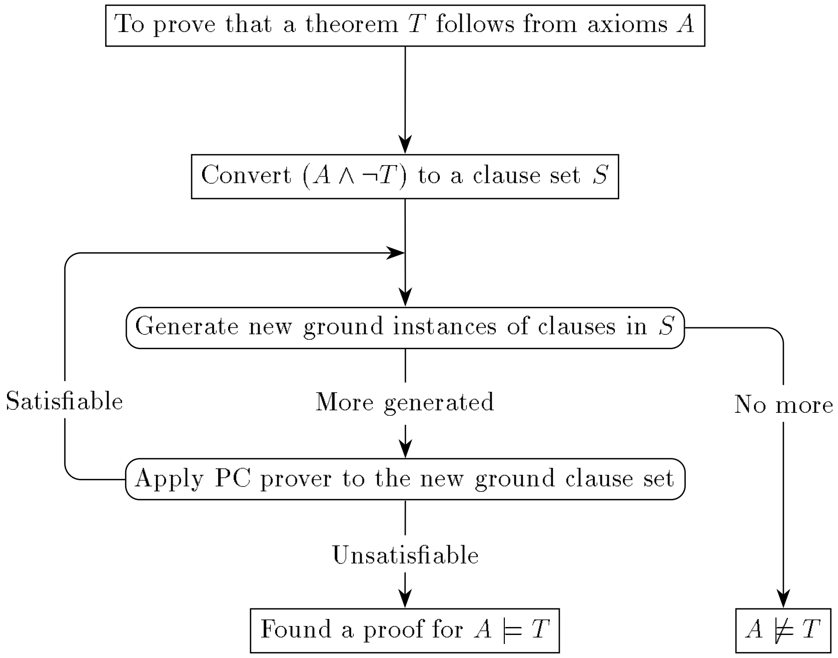


Figure 2.4: A typical instance-based theorem proving procedure

literals). This approach in general can be described as in Fig. 2.4.

One can view a non-ground axiom (for example, $needs(X, air)$ for "everyone needs air") as some fact stated over objects in the world. Ground instances of that axiom are just specific variable instantiation by objects in the world. For example, we can have $needs(john, air)$, $needs(mary, air)$, and so on. Instance-based theorem proving is based on the idea that (according to Herbrand's Theorem) a set of clauses (axioms and negated theorem) is contradictory if and only if there are some ground instances (specific instantiations) of those clauses that are contradictory. One important issue is to generate such unsatisfiable ground clause set as early as possible.

Though a simple concept, instance-based theorem proving is difficult in practice. Gilmore [Gil60] implemented a theorem prover based on a direct application of Herbrand's Theorem. His prover generated ground instances periodically by the Herbrand base enumeration and applied a propositional calculus prover for unsatisfiability test. However, due to the inefficiency of his propositional calculus prover and the huge Herbrand bases for most problems, his prover could only prove a few simple problems in first-order logic. Gilmore's propositional calculus prover was later considerably improved by Davis and Putnam [DP60] (see Section 2.3).

Lee and Plaisted [Lee90, LP92] recently implemented a theorem prover called `CLIN` based on Gilmore's idea but used a more efficient method called *hyper-linking* to generate instances of the input clauses. In essence, instead of enumerating the Herbrand base, hyper-linking uses unification to generate instances (not necessarily ground) of the input clauses. Instances are explicitly grounded and tested for unsatisfiability using a PC prover (as described in Section 2.3). Many favorable results show that hyper-linking is a promising strategy and, unlike resolution, it does not combine literals from different clauses and does not have a combinatorial explosion in the search space. In Section 2.6 we will discuss hyper-linking is more detail.

Our work on semantic hyper-linking follows the basic ideas of hyper-linking and further reduces the number of instances with the help of an input semantics. Moreover, semantic hyper-linking is slightly different and works in a model searching fashion. And only ground instances are generated so explicit grounding in not needed.

2.5 Semantically Guided Theorem Proving

Most current theorem provers apply inference rules based on syntactic features of the input clauses. For example, the resolution procedure looks for complementary literals to resolve. Some refinements on existing proof techniques use human knowledge–but only on the *proofs*, rather than the *theorems* themselves. They are all syntax oriented; they basically prepare the input clauses in such a way that the syntactically guided provers can find the proofs easily.

For example, *predicate replacement* by Lee and Plaisted [Lee90, LP92] has been used to prove many hard problems. The idea is to transform the input clauses into *replace rules* and only hyper-link some particular literals (the *distinguished literals*) of a replacement rule to generate instances. Another example is the *set of support* strategy used in both resolution [WRC65]. A subset T of a clause set S is called a *set of support* of S if $S - T$ is satisfiable. A *set-of-support resolution* is a resolution of two clauses such that at least one of them depends on T. The idea is to avoid applying resolution on the axioms, which are often satisfiable and generate a lot of resolvents.

Lee and Plaisted also implemented similar techniques in hyper-linking [Lee90].

We argue that *semantics*, namely the meanings of the input clauses, should also be used. In particular, a model (or example) for the axioms and the theorem should be used to guide the proof search.

Definition 45. A *natural semantics* is a structure that is a model of the general axioms contained in the theorem. A *trivial semantics* is a structure in which the true literals are exactly the positive (or negative) ones. We will use the terms semantics and structures interchangeably whenever there is no confusion.

This is based on the observation that proofs should rely on the meaning of the relevant concepts, not on the syntactic form used to represent the concepts [Wos88]. In addition, for human mathematicians, the use of examples (or natural semantics) is as important as the deductive abilities. In this regard, semantics also allows the provers to behave like human mathematicians. Semantics provides the knowledge for the theorem; a *semantically guided theorem prover* uses semantics to guide the search for proofs. When using a semantically guided theorem prover, one needs to know the meanings of the theorem, but does not need to know how the proof develops.

Gelernter was one of the early researchers to use semantics in theorem provers. His geometry machine [Gel59] used diagrams to help prove theorems in elementary Euclidean geometry. The diagrams were used in a heuristic way; the prover used backward chaining and rejected the unachievable subgoals not valid in the diagrams. Gelernter's system is only complete for Horn theorems.

Reiter [Rei73] presented a semantically guided deductive system using natural deduction. Semantics is used as domain-dependent heuristic in a deductive system with 12 inference rules. This is the first work to use semantics on natural deduction style theorem provers. Reiter's prover is incomplete.

Ballantyne and Bledsoe [BB82] used counterexamples to speed up proof discovery in various theorems. They used counterexamples to eliminate false subgoals during the course of the proof. The idea was similar to Gelernter's.

Slagle [Sla67] presented *semantic resolution* to use semantics in resolution. Semantic resolution is basically an extension of hyperresolution to any arbitrary structure I to use semantically satisfiable and unsatisfiable clauses instead of positive and negative clauses. Luckham [Luc68] independently proposed a refinement similar to semantic resolution. Given a semantic structure I, the binary resolution is restricted to those involving at least one clause *not* satisfied by I. Luckham's refinement is an extension of P1-resolution to arbitrary structures. The difference between Luckham's refinement and semantic resolution is analogous to that between P1-resolution and hyperresolution. Both methods have been proved complete. Henschen [Hen76] discussed the use of semantics in resolution on Horn sets.

Wang used semantics [Wan85] in his hierarchical deduction theorem prover HD-PROVER [WB87]. He introduced an *interpretation normal form* (INF) in designing examples as semantics. INF helps determine the semantics for Skolem functions. A heuristic search strategy called *partial set of support* (PSS) based on syntactic features is important in HD-PROVER to get most proofs.

Nie and Plaisted [NP90] [Nie89, Chapter 7] implemented a complete semantically backward chaining theorem prover. It is a semantic refinement of the modified problem reduction format by Plaisted [Pla82, Pla88]. It is a natural goal-subgoal system and, similar to most previous methods, unreachable subgoals are discarded. It can be seen as a generalization of Gelernter's prover to first order logic.

Sanford [San80] proposes a *hereditary lock resolution* to combine Luckham's model strategy [Luc68] and Boyer's locking resolution [Boy70]. The system is complete.

Model construction has also become an important topic recently. Instead of searching for refutation, the prover constructs models until none can be found. Winker and Wos [Win82, WW84] generated finite models and solved several open problems in ternary Boolean algebra.

Recently Slaney and Lusk [SL92] discuss an approach called *dynamic semantic resolution*, which combines model finding and the resolution procedure. They search for a finite model that satisfies the retained clauses during the proof process. They use a model-finding procedure to actually find a model dynamically. Their technique only applies to finite models.

Caferra and Zabel [CZ92] proposed a method to simultaneously search for refutation and models in resolution style prover. The model construction is done by using equations as constraints on application of resolution.

In all previous work, it is not clear how to test if a non-ground literal is satisfied by the semantics. Such a test is easy if the semantics have a finite domain of objects because it can be done by replacing variables by all possible values in the domain. When the domain is infinite, the test is not trivial and the previous method for finite domains is too loose (giving a satisfiable result when it is unsatisfiable) and incomplete. Special decision procedures, if applicable, are needed. In Section 6, we will discuss an approach to address this problem.

Semantics can also be built in the theorem prover as special inference rules or decision procedures. For example, paramodulation [RW69] and demodulation [WRCS67] are both special inference rules for equalities in resolution. Since equality is commonly used and is often the object of theorem proving, it is natural to put "hard-coded" inference rules in theorem provers to handle equalities.

Another example is Str∔ve by Hines [BH80, Hin92] which has special inference rules—*variable elimination* and *chaining*—for real linear inequalities. The special decision procedures are used to remove variables; a fast ground inequality prover is used to prove the ground literals after variables have all been removed. Our prover and Str∔ve have similar philosophy: reduce the first order problems into propositional cases. However, our prover does not combine literals from different clauses except in rough resolutions (see Section 5).

2.6 Hyper Linking

Hyper-linking (also known as *clause linking*) is a refutational clause-form theorem proving technique of first-order predicate calculus. Hyper-linking is an instance-based method which, unlike resolution, generates instances of the input clauses and reduces the problem to the propositional calculus (PC) by applying a PC prover on the ground instance set. Although in spirit hyper-linking is similar to early methods of Davis [Dav63], Gilmore [Gil60] and Prawitz [PPV60], it is much more efficient and allows various enhancements.

In this section we briefly describe hyper-linking, mainly focusing on the motivation and the main method. More detailed discussions on its enhancements and related search topics, can be found in [Lee90].

2.6.1 Motivation and Definitions

To appreciate hyper-linking, it is better to first examine the motivation and some examples. Some important definitions will also be given along with the discussion.

Hyper-linking is an instance-based technique which searches for an unsatisfiable set of ground instances. It would be useful to investigate some properties of the unsatisfiable ground set and try to answer the question "What makes this ground set unsatisfiable?"

First, let us see an example. Suppose we have a clause set S containing the following clauses:

$C1: \ \{ p(a) \}$

$C2: \ \{ \neg q(X), r(a, X) \}$

$C3: \ \{ \neg p(Y), \neg s(Z), \neg r(Y, Z) \}$

$C4: \ \{ q(b) \}$

$C5: \ \{ s(b) \}$

This clause set is unsatisfiable, which can be proved (according to Herbrand's theorem) by a PC prover on the following set S_{ground} of ground instances of S:

$C1: \ \{ p(a) \}$

$C2': \ \{ \neg q(b), r(a, b) \}$

$C3': \ \{ \neg p(a), \neg s(b), \neg r(a, b) \}$

$C4: \ \{ q(b) \}$

$C5: \ \{ s(b) \}$

$C2'$ is a ground instance of $C2$ with substitution $\{ Y \leftarrow a, Z \leftarrow b \}$, and $C3'$ is a ground instance of $C3$ with substitution $\{ X \leftarrow b \}$.

Now let us look at the ground set S_{ground} closely to examine some of its features. First of all, if any clause is taken out of S_{ground}, it becomes satisfiable. Unsatisfiable sets such as S_{ground} are minimal in retaining the unsatisfiability.

Definition 46. An unsatisfiable set S is *minimal* if S has no unsatisfiable proper subset.

Example 13. Consider clause set $S = \{ \{ \neg p, q \}, \{ p \}, \{ \neg q \}, \{ r, t \}, \{ t, p \} \}$. S is unsatisfiable but not minimal since one of its proper subset $S_g = \{ \{ \neg p, q \}, \{ p \}, \{ \neg q \} \}$ is unsatisfiable. S_g is minimal since every proper subset of S_g is satisfiable. □

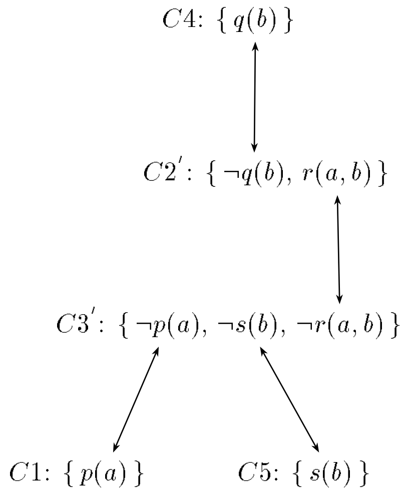


Figure 2.5: $S_{ground} = \{C1, C2', C3', C4, C5\}$ is fully linked

We also observe that for each literal L in S_{ground} there exists a complementary, $\neg L$, in some other clause. This can be illustrated by Fig. 2.5; complementary pairs are shown as linked and S_{ground} is a fully linked clause set. We can generalize this phenomenon to first-order case (which will be discussed in more detail later) and formally define as follows:

Definition 47. Given a set S of clauses, a *link* in S is a pair (L, M) of literals such that L and M appear in clauses in S and $L\theta = \neg M\theta$ for some mgu θ. L and M are said to be linked. A clause C is *fully linked* if every literal is linked.

Intuitively, it makes sense that S_{ground}, being unsatisfiable, is fully linked–every literal in the unsatisfiable set S_{ground} should be refuted by some other literal since S_{ground} denotes a contradiction. This is indeed true for any minimal unsatisfiable ground set and we can present this important fact formally in the following theorem.

Theorem 48. *If an unsatisfiable set S of ground clauses is minimal, then S is fully linked.*

Proof. Suppose the opposite: there exists a literal L that does not have a complementary literal in S. We can show that S is satisfiable by finding a model M for S

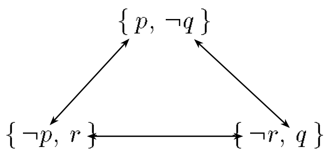


Figure 2.6: A fully-linked satisfiable set

as follows: Assigning L true will not falsify S since there exists no $\neg L$ elsewhere in S. So the model M contains at least L. Suppose L appears in clause C. Then M at least subsumes C (removing C from S). Since S is a minimally unsatisfiable set, $S - \{ C \}$ is satisfiable and the rest of M can be constructed for $S - \{ C \}$. Thus S is satisfiable with a model M. But this causes contradiction and the theorem is proved. $\square$

Remark. Note, however, that a fully linked ground set is not necessarily unsatisfiable. For example, consider the clause set $\{ \{ p, \neg q \}, \{ \neg p, r \}, \{ q, \neg r \} \}$. It is fully linked (as shown in Fig. 2.6) but it is satisfiable (with, for example, a model $\{ p, q, r \}$). This is related to the *cycle* concept in [And76].

The notion of "link" in hyper-linking is closely related to the *link conjunct method* described in [Dav63]. It is also related to the *mate* in [And76]: The "fully-linked" feature in the first-order case can be defined similarly based on, for example, the *simple* derivation. Also Theorem 48 is related to the first condition of an *acceptable* mating.

In general hyper-linking can also be seen as a partial realization of refutation by mating in the instance-based theorem proving paradigm. One major difference is that link information is not kept during hyper-linking.

After examining the above features of an unsatisfiable ground set, we will see how hyper-linking works based on those features. Recall that Herbrand's theorem provides a direct approach for instance-based theorem proving which aims at generating an unsatisfiable set of ground instances of the input clauses. The "fully-linked" property of an unsatisfiable set, especially Theorem 48, gives further direction in generating instances using unification, instead of enumeration of the Herbrand base such as Gilmore's method, to find links. Also, unification provides a means to generate instances in a more restricted manner than Gilmore's method.

Hyper-linking follows the above two directions and works toward generating a fully linked set of ground instances. The basic idea is to generate instances in which every literal has a potential complement elsewhere–this is achieved by generating fully linked instances. This is better than Gilmore's naive method in that many instances are not generated due to not being fully linked. In the next section we will discuss the hyper-linking technique in more detail.

2.6.2 The Method

Hyper-linking aims at generating a minimal unsatisfiable ground instance set by generating fully linked instances.

Two major parts of the hyper-linking procedure are:

1. Instance generation by unification: each literal of any generated instance should have a potential complement. This is based on the observation of the full-link feature of minimal unsatisfiable ground sets. Fully linked clauses are generated by *hyper-linking* using unification.

2. Satisfiability check by a PC prover: reduce the first-order instance set S to a ground set $S_\$$ by replacing all variables by a new constant \$ (\$ not appearing in the set). Then a PC prover is used to detect the satisfiability of $S_\$$. If the explicitly grounded set $S_\$$ is unsatisfiable, then, according to Herbrand's theorem, S is unsatisfiable and a proof for the original clause set is found.

Hyper-linking theorem proving basically carries out these two steps until a proof is found (by an unsatisfiable set). These two steps constitute a *round* of hyper-linking. Figure 2.7 shows the basic steps of the hyper-linking procedure.

One major advantage of hyper-linking is in the generation of instances. This is achieved through unification using *hyper-linking*. Hyper-linking[4] computes links (recall Definition 47) for all the literals in a clause. Since hyper-linking reduces

[4]The *hyper* in hyper-linking has an analogous meaning to that in hyperresolution—both indicate simultaneous operations.

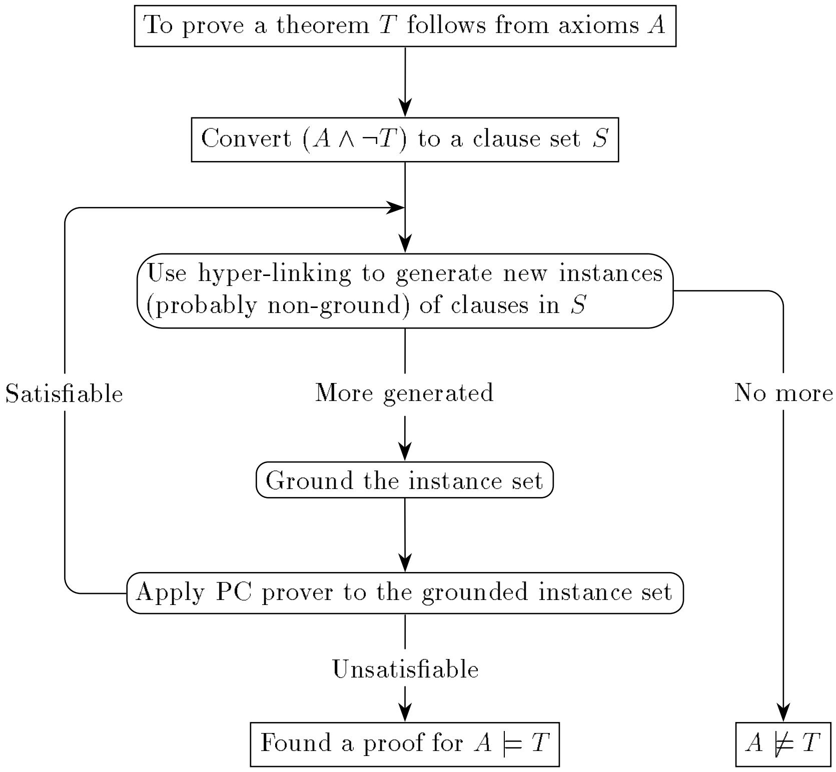
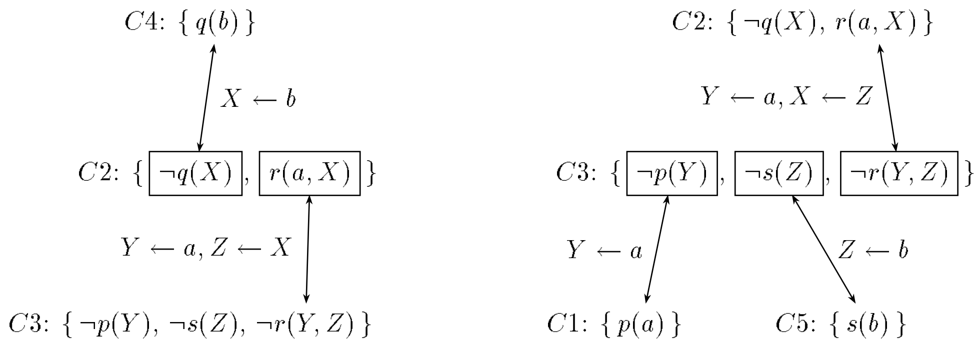


Figure 2.7: The hyper-linking procedure



(a) $C2^{'} = \{ \neg q(b), r(a, b) \}$ generated with $C2$ as nucleus

(b) $C3^{'} = \{\mathrm{p(a)}, s(b), \neg r(a, b)\}$ generated with $C3$ as nucleus

Figure 2.8: Hyper-linking operations with $C2$ and $C3$ as nuclei

the first-order problem to that in the propositional calculus, it aims at generating a fully linked ground instances set which might lead to a proof (unsatisfiability). This is achieved by hyper-link operations to generate instances of the input clauses by applying the substitutions associated with links.

Definition 49. A *hyper-link operation* generates an *instance* $C\Theta$ of a clause C and for every literal L_i in C, there exists a literal M_i and a link with M_i through Θ. C is called the *nucleus* and M_i are the *electrons*.

The instances generated by hyper-link operations are in general non-ground and can be used in further hyper-link operations. To check satisfiability, a PC prover is applied to the explicitly grounded clause set by replacing all variables by a new constant '\$'. The explicit grounding operation is to obtain a specific ground instance for each clause containing variables. If such a ground set is unsatisfiable, then the original clause set is unsatisfiable according to Herbrand's Theorem.

Example 14. Let us consider the clause set S in Section 2.6.1 (see Page 39). With $C2$ as nucleus, hyper-linking generates (as shown in Fig. 2.8(a)) the instance

$$C2^{'} = C2\theta_1 = \{ \neg q(b), r(a, b) \}$$

with $\theta_1 = \{ X \leftarrow b, Y \leftarrow a, Z \leftarrow X \}$.

With $C3$ as nucleus, hyper-linking generates (as shown in Fig. 2.8(b)) the instance

$$C3' = C3\theta_2 = \{ \neg p(a), \neg s(b), \neg r(a, b) \}$$

with $\theta_2 = \{ Y \leftarrow a, Z \leftarrow b, X \leftarrow Z \}$. After $C2'$ and $C3'$ are generated in this round, we obtain the clause set $\{ C1, C2, C3, C4, C5, C2', C3' \}$, in which $C2$ and $C3$ are non-ground and need to be explicitly grounded before the PC prover is applied. $C2$ is grounded to $\{ \neg q(\$), r(a, \$) \}$ and $C3$ is grounded to $\{ \neg p(\$), \neg s(\$), \neg r(\$, \$) \}$. The resulting ground set is unsatisfiable as shown in Fig. 2.5 and a proof is found. □

In addition to its simplicity, hyper-linking also has another advantage over previous instance-based theorem proving techniques: it greatly reduces the number of generated ground instances by the restriction naturally imposed by hyper-link operations. A hyper-link operation generates an instance, say $C\theta$, only after assuring that *each* literal in $C\theta$ has a potential link. This is clearly better than, for example, enumerating the Herbrand base.

Example 15. Suppose we use the Herbrand base enumeration in Example 14, we would have generated the following ground instances from $C3$:

$\{ \neg p(a), \neg s(a), \neg r(a, a) \}$

$\{ \neg p(a), \neg s(b), \neg r(a, b) \}$ $=C3'$

$\{ \neg p(b), \neg s(a), \neg r(b, a) \}$

$\{ \neg p(b), \neg s(b), \neg r(b, b) \}$

Note that now we have to generate four instances while hyper-linking only generates $C3'$. □

Other refinements to hyper-linking include small proof checking which looks for hyper-linking in which all of the electrons are either existing unit clauses or derivable by one UR resolution step; set of support strategy which achieves similar effect as that for resolution; a priority strategy which prefers smaller (in terms of the priority) clauses; and predicate replacement which allows hyper-linking to proceed in a definition expansion fashion and helps to prove many difficult problems.

More detailed discussions can be found in [Lee90, LP92].

Chapter 3

Semantic Hyper-Linking

3.1 Introduction

In this chapter, we describe *semantic hyper-linking* to use semantics with hyper-linking to prove hard theorems. Semantics about the theorem and the axioms is given as input; the theorem prover applies inference rules based on both the semantic and syntactic features of the input data. With no other human guidance, our prover can obtain proofs for hard theorems that other powerful provers (for example, Lee's prover `CLIN` or `Otter` [McC90]) cannot.

Intuitively, semantic hyper-linking can be seen as a model searching process—a process searching for models of the input clauses. To understand how this model searching process works, it is helpful to look at one motivating example first. This example is mainly to illustrate the model searching idea although it is unlikely to really happen.

Example 16. Suppose a student John tries to prove a theorem T in geometry with a given set A of axioms. He first draws a diagram I_1 compliant with the axioms. Then he starts to prove by contradiction, namely, first assuming T to be false and hoping to derive contradiction from $\neg T$ and A. His approach is, instead of generating lemmas, to generate specific instances (ground instances) of axioms A or negated theorem $\neg T$. For example, if John has the following axiom:

$$\forall x \forall y (\overline{xy} = \overline{yx}) \textit{ distance from } x \textit{ to } y \textit{ equals that from } y \textit{ to } x$$

and points a and b in the diagram, John could generate the "ground instances" $\overline{ab} = \overline{ba}$ and $\overline{ba} = \overline{ab}$.

John uses the diagram to help focus on deriving related ground instances: Since the goal is to derive a contradiction, John concentrates on deriving ground instances that contradict the diagram he uses. To start, John derives ground instances[1] that contradict I_1. Now, I_1 is not a right diagram for the generated instances, so he has to draw a new diagram I_2 to satisfy every instance he has derived so far (from A and $\neg T$). Further derived ground instances contradict I_2 again so he needs to use another new diagram I_3. By deriving ground instances that contradict the diagram being used, John changes the diagrams repeatedly to obtain I_4, I_5, $\ldots$, until he finally derives a contradiction among the generated instances (for simplicity, let us assume such a contradiction finding method is known to John), at which point a proof has been found. $\square$

In the example, John has obtained a proof by applying a model searching process. Each diagram I_i is a model for the ground instances he has generated so far; with I_i, John tries to generate ground instances falsifying I_i. John found a proof when all the instances he has generated cause a contradiction. Although such proof searching method seems unusual to a human, it is the basic searching philosophy of semantic hyper-linking. In the following sections, we will discuss the method in more detail.

3.2 Motivation and Definitions

In this section, we use an example to give the motivation of our method to apply the aforementioned model searching process in refutation theorem proving. With the discussion, we also give several important definitions pertinent to our method.

Let S be the set of clauses $\{P(a), \neg P(x) \vee P(f(x)), \neg P(f(f(a)))\}$. The Herbrand base is then $\{P(a), P(f(a)), P(f(f(a))), P(f(f(f(a)))), ...\}$. A (Herbrand) structure I can then be viewed as a mapping from the Herbrand base to $\{\mathbf{T}, \mathbf{F}\}$. For each literal L, either $I \models L$ or $I \not\models L$. Recall that in the former case we say that I *satisfies* L (L is true in I) and in the latter case, I *does not satisfy* L (L is false in I). Also, as

[1]Most likely the first instances are from the negated theorem if a proper diagram is used.

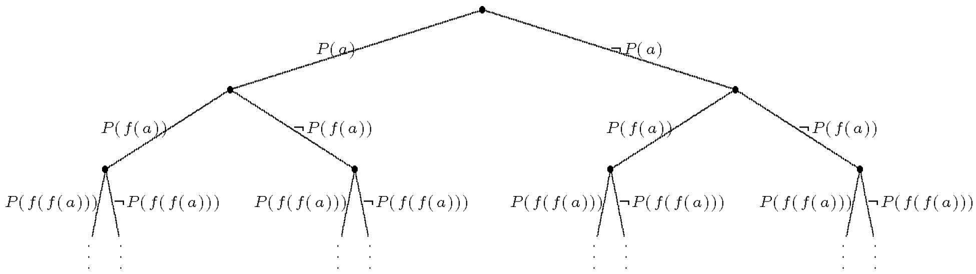


Figure 3.1: A semantic tree for the clause set $\{P(a), \neg P(x) \vee P(f(x)), \neg P(f(f(a)))\}$.

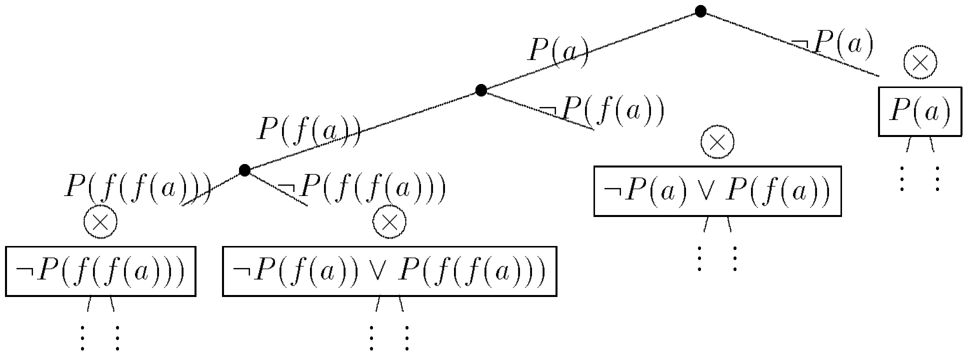


Figure 3.2: The contradiction nodes and associated ground instances in Fig. 3.1 after model finding.

mentioned in Section 2.4.2, we can represent the set of all such Herbrand structures by an infinite semantic tree, as shown in Fig. 3.1.

In this binary tree, infinite paths correspond to structures. For example, the leftmost path corresponds to the structure $\{P(a), P(f(a)), P(f(f(a))), ...\}$.

Semantic hyper-linking can be viewed as a systematic search for a model in this tree. If this search fails, then we have shown that S is unsatisfiable and a proof has been found. Supposing that we always take the leftmost choice first, and backtrack when a contradiction is found, we obtain the tree in Fig. 3.2.

The circled nodes are contradiction nodes and for each such node we have an associated ground instance of an input clause that is not satisfied by the path to the node. The set of ground instances obtained in this way is then $\{\neg P(f(f(a))), \neg P(f(a)) \vee P(f(f(a))), \neg P(a) \vee P(f(a)), P(a)\}$. This set of clauses is unsatisfiable, and this can be shown by a PC procedure.

We can view this search process in the following way: We generate a sequence

I_1, I_2, I_3, I_4 of four structures:

$$
\begin{aligned}
I_1 &= \{\, P(a), P(f(a)), P(f(f(a))), P(f(f(f(a)))), \ldots \} \\
I_2 &= \{\, P(a), P(f(a)), \neg P(f(f(a))), P(f(f(f(a)))), \ldots \} \\
I_3 &= \{\, P(a), \neg P(f(a)), P(f(f(a))), P(f(f(f(a)))), \ldots \} \\
I_4 &= \{\, \neg P(a), P(f(a)), P(f(f(a))), P(f(f(f(a)))), \ldots \}
\end{aligned}
$$

We note that $I_1 \not\models \neg P(f(f(a)))$, $I_2 \not\models \neg P(f(a)) \vee P(f(f(a)))$, $I_3 \not\models \neg P(a) \vee P(f(a))$, $I_4 \not\models P(a)$, each of which explains why the search backtracks on individual contradiction nodes.

We represent these (infinite) structures I_i by a finite *explicit part* and an infinite *implicit part*, as follows: Let I be the initial structure I_1. Then I_2 is $I[\{\, \neg P(f(f(a)))\,\}]$, I_3 is $I[\{\, \neg P(f(a))\,\}]$, and I_4 is $I[\{\, \neg P(a)\,\}]$. We now give a formal definition for $I[\mathcal{L}]$.

Definition 50. Let $\mathcal{L}$ be a set of ground literals, and $\overline{\mathcal{L}}$ be the set of complements of literals in $\mathcal{L}$. $\mathcal{L}^*= \mathcal{L} \cup \overline{\mathcal{L}}$. □

Definition 51. For a structure I and a set $\mathcal{L}$ of ground literals, a new structure $I[\mathcal{L}]$ is defined as follows (when expressed as a set of all literals that are true in this structure):

$$I[\mathcal{L}] = (I - \overline{\mathcal{L}}) \cup \mathcal{L}$$

In other words, $I[\mathcal{L}]$ is the same as I except that the literals in $\mathcal{L}$ are satisfied by $I[\mathcal{L}]$ (for all L in $\mathcal{L}$, $I \not\models L$). □

The explicit part of $I[\mathcal{L}]$ is $\mathcal{L}$, which is represented declaratively by a set of literals. The implicit part is I which is represented procedurally, by procedures for computing truth values in I and, in some cases, testing satisfiability in I. Thus each structure I_i (except for the initial one) generated during the search for models, needs not be explicitly constructed. And throughout the whole process, only the specification of the initial structure, I, is needed as the implicit part of any $I[\mathcal{L}]$.

Definition 52. Given a structure represented as $I[\mathcal{L}]$, the literals in $\mathcal{L}$ are called *eligible literals*.

In the following discussion, it will become clear that eligible literals are those eligible for generating one or more ground instances C such that $I[\mathcal{L}] \not\models C$.

Based on the model searching process described in Section 3.1, semantic hyper-linking generates a sequence I_j of structures, and for each I_j, represented as $I[\mathcal{L}]$, we find one (or more) ground clauses C such that $I_j \not\models C$. Such C causes the model searching in the semantic tree to backtrack to find a new structure I_{j+1}. A proof is found after no more structures can be generated and the set of ground clauses is unsatisfiable (or, as in Fig. 3.2, the semantic tree is closed by contradiction nodes in all branches). Note how the model searching idea (introduced in Section 3.1) applies here. This is the major motivation of semantic hyper-linking.

3.3 Semantic Hyper-Linking

With a structure I_i represented as $I[\mathcal{L}]$, the key idea of semantic hyper-linking is to find ground instances that are not satisfied by $I[\mathcal{L}]$. The fundamental problems are: Given I and finite set $\mathcal{L}$ of literals and clause C, does $I[\mathcal{L}] \models C$, and if not, find a ground instance $C\Theta$ of C such that $I[\mathcal{L}] \not\models C\Theta$. The following is an important observation.

Remark. If $I[\mathcal{L}] \not\models D$ and D is a ground clause then for all L in D, $I[\mathcal{L}] \not\models L$. We can express D as $D_1 \cup D_2$ where the D_i are disjoint and $D_1 \subseteq \mathcal{L}^*$ and $D_2 \cap \mathcal{L}^* = \phi$. Then $D_1 \subseteq \overline{\mathcal{L}}$ and $I \not\models D_2$. □

If C is a non-ground clause, then we are looking for such ground instances D of C. So the idea is to express C as $C_1 \cup C_2$ where $C_1 \cap C_2 = \phi$, and then find a substitution α such that $C_1\alpha \subseteq \overline{\mathcal{L}}$ and find a substitution β such that $C_2\alpha\beta$ is ground and $I \not\models C_2\alpha\beta$. Also we need to have that $C_2\alpha\beta \cap \mathcal{L}^* = \phi$. Now, α can be found just by a subsumption test, since we are essentially testing if C_1 subsumes $\overline{\mathcal{L}}$; such tests are well-known. The method we use is to look for α by hyper-linking C_1 literals with those in $\mathcal{L}$. Then we search for β by a process of exhaustive enumeration.

Example 17. Here is an example of the method of finding α and β. Let I be $\{P(a), P(f(a)), P(f(f(a))), P(f(f(f(a)))), ...\}$ as before. Let $\mathcal{L}$ be $\{\neg P(f(f(a)))\}$. Let the clause C be $\{\neg P(x), P(f(x))\}$. Suppose we choose C_1 as $\{P(f(x))\}$ and C_2 as $\{\neg P(x)\}$. Then we seek α such that $C_1\alpha \subseteq \overline{\mathcal{L}}$. For this we take α to be $\{x \leftarrow f(a)\}$. In this case $C_2\alpha$ is ground and $I \not\models C_2\alpha$. So we choose β as the identity substitution, and obtain $C\alpha\beta$ as $\{\neg P(f(a)), P(f(f(a)))\}$. One can check that $I[\mathcal{L}] \not\models C\alpha\beta$. We note that in this case it is not necessary to use exhaustive enumeration since $C_2\alpha$ is already a ground clause. This is an interesting special case that is much faster than the general case, and we make more use of it (to be discussed in Section 3.4 and Chapter 4). □

Definition 53. The process of generating $C_1\alpha$ is called *partial hyper-linking*, the instance $C\alpha$ is a *partial hyper-link*. The *ground instance generation* process on partial hyper-links generates $C_2\alpha\beta$. After partial hyper-linking and ground instance generation, a ground instance $C\alpha\beta$ is obtained and $I[\mathcal{L}] \not\models C\alpha\beta$. □

Therefore, in semantic hyper-linking, instances are generated by partial hyper-linking and ground instance generation. Partial hyper-linking is finite and deterministic for a given finite $\mathcal{L}$ and a clause C. Ground instance generation, however, has to search through a possibly infinite number of ground instances of a given $C_2\alpha$, which might be computationally expensive. And the number of instances false in the input I might also be infinite.

The large search space of the first problem can be reduced by using a decidable structure.

Definition 54. A structure I is said to be *decidable* if, given a clause C possibly containing variables, it is decidable whether $I \models C$. The test for $I \models C$ is called a *semantic check*.

With such a decidable I, we can then test if $I \models C_2\alpha$. If so, we know that no β exists because every instance of $C_2\alpha$ is satisfied by I. Otherwise, we proceed to do exhaustive enumeration to find β. Since there might be many possible α and C_2 for a clause C and $I[\mathcal{L}]$, such checks greatly reduce the number of $C_2\alpha$ and make the exhaustive enumeration much faster. Some examples of using a decidable I are discussed in Chapter 6.

The second problem with ground instance generation, due to choosing among infinitely many instances false in I, is related to the completeness of semantic hyper-linking.

Definition 55. A partial ordering $\prec$ is *well-founded* if there are no infinite sequences x_i of elements such that $x_{i+1} \prec x_i$ for all $i \geq 0$.

It turns out that given a well-founded ordering $\prec$ on ground clauses, we only need to generate those smallest $C_2\alpha\beta$ false in I. The completeness is maintained and the number of instances generated is greatly reduced. We will give more details on completeness in Section 3.5.

For a structure I_j, after one (or more) ground clauses C are generated such that $I_j \not\models C$, a new structure I_{j+1} is constructed by building a model for all the ground clauses already generated. Since the ground clause set is finite, a finite model M_{j+1} can be found. This is equivalent to finding a finite path in the semantic tree (as shown in Section 3.2) from the root to a *model node* (the path from the root to this node corresponds to the ground model). More importantly, none of the structures $I_1, \ldots, I_j$ satisfies the ground clause set. In other words, the paths closed by contradiction nodes need not be searched again. This allows the model searching in the semantic tree to be carried out in an *incremental* manner. Thus we call it *incremental model finding* [CP93a, CP93b]. From M_{j+1}, a finite subset $\mathcal{L}_{j+1}$ of ground literals can be constructed such that $I_{j+1} = I[\mathcal{L}_{j+1}]$. Note that $\mathcal{L}_{j+1} \subseteq M_{j+1} \subseteq I_{j+1}$ and I_{j+1} satisfies all ground clauses so far. We will discuss incremental model finding in more details in Chapter 4. Each loop in which a new structure is found, is called a *round*. Figure 3.3 shows a conceptual diagram illustrating how semantic hyper-linking conducts model searching. The overall semantic hyper-linking algorithm is shown in Fig. 3.4.

3.4 Finding Better Models

Recall that a process of exhaustive enumeration is needed to obtain β. Although we can use decidable structures to filter out valid $C_2\alpha$ and only need to generate the smallest $C_2\alpha\beta$ (with respect to a given well-founded ordering $\prec$), this is still expensive. So we try to avoid it whenever possible. Thus we make this test for contradicting

Given input semantics I and input clause set S

Round 1: $I_1 = I \xrightarrow[\text{Generation}]{\text{Instance}} D_1 \xrightarrow{\text{PC Prover}} M_2 \models D_1$

$\mathcal{L}_2$

Round 2: $I_2 = I[\mathcal{L}_2] \xrightarrow[\text{Generation}]{\text{Instance}} D_2 \xrightarrow{\text{PC Prover}} M_3 \models (D_1 \cup D_2)$

$\mathcal{L}_3$

Round 3: $I_3 = I[\mathcal{L}_3] \xrightarrow[\text{Generation}]{\text{Instance}} D_3 \xrightarrow{\text{PC Prover}} M_4 \models (D_1 \cup D_2 \cup D_3)$

$\vdots$

Round n: $I_n = I[\mathcal{L}_n] \xrightarrow[\text{Generation}]{\text{Instance}} D_n \xrightarrow{\text{PC Prover}} M_{n+1} \models (D_1 \cup D_2 \ldots \cup D_n)$

$\vdots$

- $\prec$: an ordering defined on ground clauses and for any C there exists finitely many D such that $D \prec C$
- D_i: set of smallest (w.r.t. $\prec$) ground instances of clauses in S such that for all $C \in D_i, I \not\models C$
- $\mathcal{L}_i$: the largest subset of M_i such that for all $L \in \mathcal{L}_i$, $I \not\models L$
- $\mathcal{L}_i \subseteq M_i \subseteq I_i = I[\mathcal{L}_i]$
- A proof is found if the ground instance set $\bigcup_i D_i$ is unsatisfiable

Figure 3.3: A conceptual diagram of semantic hyper-linking

```
Algorithm Semantic Hyper-Linking(S,I,≺)
input
  S: input clause set
  I: input semantics
  ≺: a well-founded ordering on ground clauses
output
  satisfiable: S is satisfiable
  unsatisfiable: S is unsatisfiable
begin
  % the ground clause set
  T={ }
  % current semantics
  I_now= I
  % each loop corresponds to a round
  loop
    if I_now ⊨ S
    then
      S is satisfiable
      return
    else
      D= set of smallest (w.r.t. ≺) ground instances of clauses in S
           such that for all C ∈ D, I_now ⊭ C
      T = T ∪ D
      if T is unsatisfiable
      then
        % Found a proof
        S is unsatisfiable
        return
      else
        % Find a model M for T
        % Find the eligible literal set ℒ from M
        I_now=a new semantics I[ℒ] such that I[ℒ] ⊨ T
  forever
end
```

Figure 3.4: Semantic hyper-linking algorithm

I_j more powerful, to include not only *direct contradictions* (that is, such ground instances C) but also *indirect contradictions* (that is, logical consequences of S that are not satisfied by I_j). This is done to avoid the enumeration involved in generating β.

Let I_j be $I[\mathcal{L}_j]$ and suppose the model for the ground clause set is M_j (refer to Fig. 3.3). Note that $\mathcal{L}_j \subseteq M_j$. For each I_j we generate a set S_j of ground instances of clauses in S without using exhaustive enumeration, and we test whether $M_j \cup S_j$ is satisfiable. (Here M_j is considered as a conjunction of literals.) If not, we know that we can backtrack and go on to the next M_j (and thus I_j) without performing the exhaustive enumeration involved in finding β. This advances the I_j in larger steps without using semantics and helps to find proofs faster. This process is especially useful in areas in which defined concepts are common, such as in set theory where set union and set intersection and other operations can be defined in terms of set membership. It turns out that such definitions correspond to clauses in which the enumeration of β is not necessary. This helps a lot in set theory and temporal logic and similar areas.

Example 18. Let the ground model M be $\{ a = e, a = b, b = c, c = d, a \neq d \}$. Suppose we have the axiom $\{x \neq y, y \neq z, x = z\}$. Then we note that this axiom implicitly contradicts M; from $a = b$ and $b = c$ we can derive $a = c$. From this and $c = d$ we can derive $a = d$. But this contradicts $a \neq d$ contained in M. Our prover searches for such implicit contradictions and has several specialized methods for finding them. □

In Chapter 4, several specialized methods are discussed in detail to check for direct contradiction (using *model filtering*) or indirect contradictions (using *model literal replacement* and *UR resolution*). They make the model finding backtrack by generating clauses containing negations of the model literals. The effect of those methods is, among many models to choose from, finding a "better" model for a given ground clause set. One major advantage is that the search space is not expanded by introducing new literals (as semantic hyper-linking does). Experimental results have shown that those methods are quite powerful and helpful to find proofs much faster.

3.5 Soundness and Completeness

Theorem 56 (Soundness). *Given a clause set S, if semantic hyper-linking finds a proof, then S is unsatisfiable.*

Proof. Since semantic hyper-linking generates ground instances of the input clauses, if an unsatisfiable ground instance set is detected by the model finding procedure, the input clause set is unsatisfiable according to Herbrand's Theorem. So semantic hyper-linking is sound. $\square$

On the issue of completeness, a well-founded complexity ordering $\prec$ has to be defined on ground clauses which satisfies the property that for any clause C, $\{ D : D \prec C \}$ is finite. For example, $\prec$ can be based on comparison of lengths when written as character strings, or $\prec$ could be lexicographical ordering between those strings. In practice, we choose $\prec$ to be the ordering of the lengths of the maximum literals, when written as character strings, in a clause.

Definition 57. Two structures I_1 and I_2 are *identical for set S of literals* if for all literals L in S, $I_1 \models L$ if and only if $I_2 \models L$. $\square$

Theorem 58 (Completeness). *If a ground instance C is generated for $I[\mathcal{L}]$ such that C is minimal in the complexity ordering $\prec$ subject to the condition that $I[\mathcal{L}] \not\models C$, semantic hyper-linking will find a proof if the input clause set is unsatisfiable.*

Proof. Suppose semantic hyper-linking is not complete. Then there is an unsatisfiable set S of clauses in which semantic hyper-linking never terminates. Thus there is an infinite sequence $I_1, I_2, \ldots$ of structures chosen, and an infinite sequence $D_1, D_2, \ldots$ of ground instances of clauses in S chosen such that $I_i \not\models D_i$ and $I_j \models D_i$ for all $i < j$ by the way I_i is chosen.

Let T be a semantic tree for S. Each branch of T corresponds to a structure for S (set of all literals attached to all links of that branch). According to Herbrand's Theorem, corresponding to T there is a closed semantic tree T' for S. T' is obtained by adding contradiction nodes to T. Each contradiction node corresponds to a ground instance of a clause in S.

We now attempt to "close" the semantic tree T based on the infinite sequence of structures and ground instances chosen by semantic hyper-linking. Since $I_i \not\models D_i$, each D_i corresponds to a contradiction node in T which closes the branch corresponding to I_i. Also, since $I_j \models D_i$ for all $i < j$, each closed branch has only one contradiction node, namely, only D_i for the corresponding I_i. Thus, according to König's Lemma, with the infinite sequences of D_i and I_i used to close branches of T, there is an infinite branch B in T that semantic hyper-linking is unable to close. Let C_B be the ground instance corresponding to the contradiction node N which closes B in T'.

Suppose the set $L = \{ L_1, \ldots, L_n \}$ contains those literals attached to the links in the path from root to N. Since $I_j \models D_i$ for all $i < j$ and $I_i \not\models D_i$ for all i, there are infinitely many structures that are identical for L and have corresponding branches closed at descendants of N (otherwise, there must be one branch that is closed by more than one contradiction node). Let $I(L)$ denote the infinite set of such structures. Note that $I_i \not\models C_B$ for all $I_i \in I(L)$. For each I_i in $I(L)$, semantic hyper-linking chooses a ground instance D_i such that $I_i \not\models D_i$ and D_i is minimal in the complexity ordering $\prec$. Since there are only finitely many ground clauses that are strictly smaller than C_B, C_B must be some D_i after a finite number of D_i are chosen. This implies that there exists one $D_k = C_B$ such that for all $j > k, I_j \not\models D_k$, namely,

$$\text{there is one } D_k \text{ such that } I_j \not\models D_k \text{ for infinitely many } I_j \tag{3.1}$$

However, we already know that for all $j > i, I_j \models D_i$ and there are infinitely many I_j. Thus,

$$\text{for each } D_i \text{ there are only finitely many } I_j \text{ such that } I_j \not\models D_i. \tag{3.2}$$

Results (3.1) and (3.2) cause contradiction. Thus the theorem is proved. □

Note that the completeness proof is independent of the input structure I. So in practice any structure, even a trivial one (for example, a structure in which the true literals are exactly the positive ones), can be used. This is especially useful since a natural semantics is not always available. We also found out that even with a trivial semantics, semantic hyper-linking still performs pretty well on some hard problems that are difficult for some other theorem provers (see Chapter 8).

In later discussions, it will also be proved that completeness is retained when semantic hyper-linking is augmented by various procedures.

In Chapter 4, several procedures are given to derive D_i such that $I_i \not\models D_i$ without using semantic hyper-linking which might be expensive. Since no new literals are introduced, from the viewpoint of model searching in a semantic tree, those procedures search for a "better" model for a ground clause set. The soundness and completeness will be discussed also.

In Chapter 5, it will be shown that the restriction to enumerate smallest literals first (with respect to $\prec$) imposes a difficulty on enumerating large literals which are needed to find the proofs—larger ones will not be chosen until all the smaller ones are already chosen. To cope with this problem, *rough resolution* is used to remove large literals so they do not have to be generated at all. Also, the soundness and completeness will be discussed.

3.6 An Example

In this section, we will walk through an example of moderate size to illustrate the major ideas of semantic hyper-linking.

This theorem asserts that, in a subgroup H of a group G, for any element $a \in H$, its inverse a^{-1} is still in H. Two predicates are used: $s(x)$ means x is in the subgroup H; $p(x, y, z)$ means $xy = z$. Function $i(x)$ denotes x^{-1}; constant e is the identity of G.

The list of axioms is as follows:

Identity: $\forall x \in G$, $ex = x$ and $xe = x$.

$C1: \quad \{\, p(e, x, x) \,\}$

$C2: \quad \{\, p(x, e, x) \,\}$

Inverses: $\forall x \in G$, $x^{-1}x = e$ and $xx^{-1} = e$.

$C3: \quad \{\, p(i(x), x, e) \,\}$

$C4: \quad \{\, p(x, i(x), e) \,\}$

Associativity: $\forall x, y, z \in G$, $(xy)z = x(yz)$ and $x(yz) = (xy)z$.

$C5: \quad \{\, \neg p(x, y, v), \neg p(y, z, w), \neg p(x, w, u), p(v, z, u) \,\}$

$C6: \quad \{\, \neg p(x, y, v), \neg p(y, z, w), \neg p(v, z, u), p(x, w, u) \,\}$

H is a subgroup: $\forall x, y \in H$, $xy^{-1} \in H$

$C7: \quad \{ \neg s(x), \neg s(y), \neg p(x, i(y), z), s(z) \}$

The theorem is $\forall x(s(x) \supset s(i(x)))$, the negation of the theorem introduces a Skolem constant b:

$C8: \quad \{ s(b) \}$

$C9: \quad \{ \neg s(i(b)) \}$

Suppose we choose the input structure I to be the additive group Z_6 of integers modulo 6, and a cyclic group <2> as the subgroup. So $G = \{ 0, 1, 2, 3, 4, 5 \}$, and $H = \{ 0, 2, 4 \}$. Then e^I is 0 and we choose b^I to be 2 in H. $s^I(x)$ means that $x \in \{ 0, 2, 4 \}$ and $i^I(x)$ is $6 - x$. It can be checked that $s(i(b))$ is satisfied by I since $b^I = 2$ and $i^I(2) = 4$ and $s^I(4)$ is true. On the other hand, $p(b, i(b), b)$ is not satisfied by I.

We list in Fig. 3.5–3.9 the search of the semantic tree by the model finding in each round. Intermediate clause sets by the case analysis of the PC prover (indicated by $\times$) are omitted. Each leaf is boxed and is either a model node, denoted by a model, or a contradiction node, denoted by $\otimes$ and a contradicted ground instance of an input clause. The complexity ordering is on the maximal literal size—denoted by the number of symbols for atoms, variables, functions and predicates that appear in the literal—of a clause.

Note that in this example, the first (left) cases are always positive literals. This is just a coincidence—the first cases should always be *true* in the input structure I (to be explained in Chapter 4). Of course, in general, true literals could also be negative.

1. At the beginning, $I_1 = I$. Note that $I_1 \not\models C9(\{ \neg s(i(b)) \})$. After a model $M_2 = \{ s(b), \neg s(i(b)) \}$ is found for the ground set $\{ C8, C9 \}$, a new structure $I_2 = I[\{ \neg s(i(b)) \}]$ is constructed. And $\neg s(i(b))$ is the only eligible literal. Figure 3.5 shows the search for M_2.

2. To find ground clauses false in I_2, we split $C7$ into $C7_a = \{ s(z) \}$ and $C7_b = \{ \neg s(x), \neg s(y), \neg p(x, i(y), z) \}$. Recall the remark from Sec. 3.3 that $C7_a\alpha_1$ is D_1 and $C7_b\alpha_1$ has ground instances D_2. Then we generate partial hyper-link $C7\alpha_1$, for $\alpha_1 = \{ z \leftarrow i(b) \}$, by linking the literal $s(z)$ of $C7$ with $\neg s(i(b))$. Then $C7_a\alpha_1 = \{ s(i(b)) \}$ and $C7_b\alpha_1$ is $\{ \neg s(x), \neg s(y), \neg p(x, i(y), i(b)) \}$. From this partial hyper-link we generate the (smallest) ground instance of $C7\alpha_1$ by

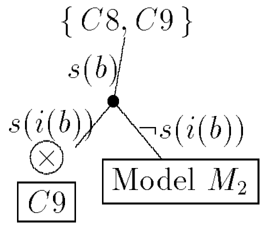


Figure 3.5: Model finding in round 1

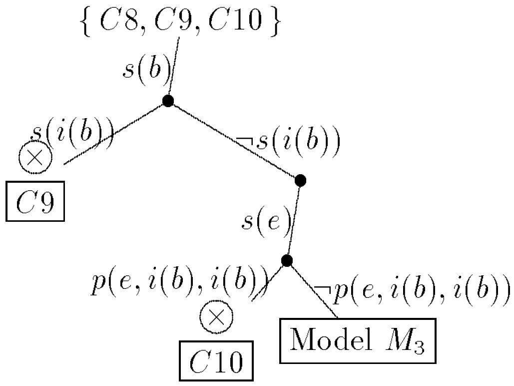


Figure 3.6: Model finding in round 2

finding $\beta_1 = \{ x \leftarrow e, y \leftarrow b \}$. We then obtain $C10 = C7\alpha_1\beta_1$:

$$C10 : \{ s(i(b)), \neg s(e), \neg s(b), \neg p(e, i(b), i(b)) \}$$

Now the ground clause set is $\{ C8, C9, C10 \}$. From this ground clause set, incremental model finding finds a model $M_3 = \{ s(b), \neg s(i(b)), s(e), \neg p(e, i(b), i(b)) \}$ and $\mathcal{L}_3 = \{ \neg s(i(b)), \neg p(e, i(b), i(b)) \}$. Thus a new structure $I_3 = I[\mathcal{L}_3] = I[\{ \neg s(i(b)), \neg p(e, i(b), i(b)) \}]$ ($\neg s(i(b))$ and $\neg p(e, i(b), i(b))$ are eligible literals) is constructed. The semantic tree is shown in Fig. 3.6.

3. In the third round, we pick $C1_1 = C1$ and $C1_2 = \{ \}$ and hyper-link $C1_1$ with $\neg p(e, i(b), i(b))$ to generate the ground instance $C11 = C1\alpha_2$, where $\alpha_2 = \{ x \leftarrow b \}$:

$$C11 : \{ p(e, i(b), i(b)) \}$$

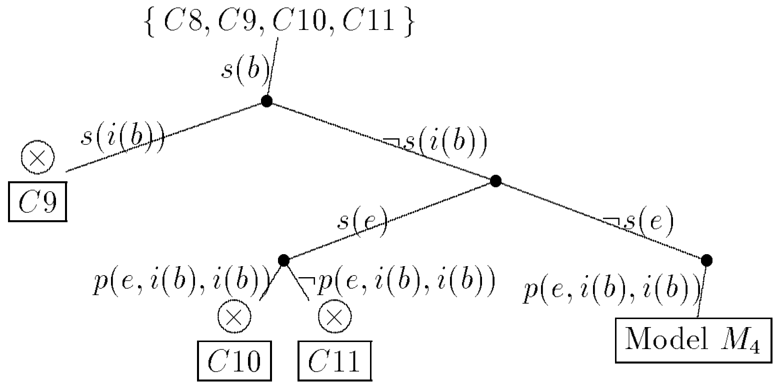


Figure 3.7: Model finding in round 3

Note that a partial hyper-link $C7\alpha_1$ is still generated in this round[2], but $C11$ is smaller than any of $C7\alpha_1$'s ground instances (except $C10$ which has been generated in round 2).

Now the ground set is $\{ C8, C9, C10, C11 \}$ and $I_3 \not\models C11$, so the model finding process backtracks and finds a new model $M_4 = \{ s(b), \neg s(i(b)), \neg s(e), p(e, i(b), i(b))\}$. From M_4, a new structure $I_4 = I[\{ \neg s(i(b)), \neg s(e) \}]$ is constructed. The semantic tree is shown in Fig. 3.7.

4. The next partial hyper-linking generates two partial hyper-links from $C7$: $C7\alpha_1$ (as shown before) and $C7\alpha_3$, obtained by linking the literal $s(z)$ of $C7$ with $\neg s(e)$, where α_3 is $\{ z \leftarrow e \}$. Thus $C7_a\alpha_3 = \{ s(e) \}$ and $C7_b\alpha_3 = \{\neg s(x), \neg s(y), \neg p(x, i(y), e)\}$. One smallest ground instance $C12 = C7\alpha_3\beta_3$, where $\beta_3 = \{ x \leftarrow b, y \leftarrow b \}$, is generated from $C7\alpha_3$:

$$C12 : \{ s(e), \neg s(b), \neg s(b), \neg p(b, i(b), e) \}$$

The clause $\{ s(e), \neg s(e), \neg s(e), \neg p(e, i(e), e) \}$ is also generated from $C7\alpha_3$ but we delete it since it is a tautology. With $C12$ added, the ground set becomes $\{ C8, C9, C10, C11, C12 \}$. Since $I_4 \not\models C12$, the model finding process backtracks and finds a new model $M_5 = \{ s(b), \neg s(i(b)), \neg s(e), p(e, i(b), i(b)),$

[2]We are not showing all possible hyper-links; we only show those that are useful for a proof.

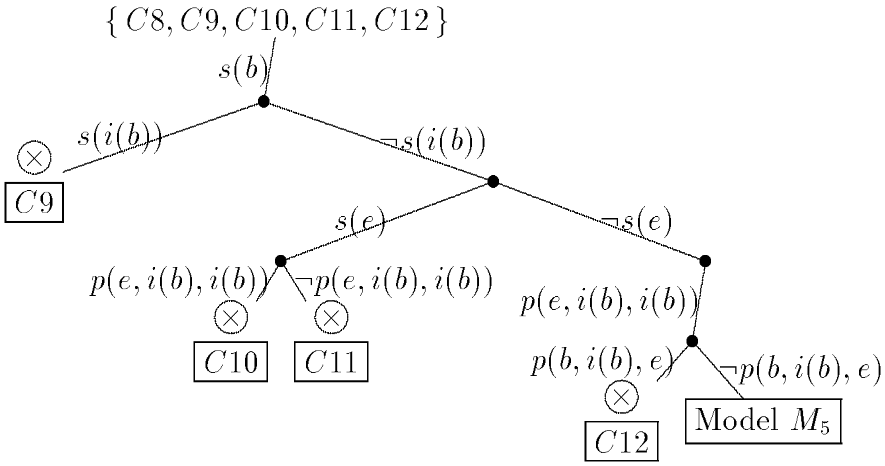


Figure 3.8: Model finding in round 4

$\neg p(b, i(b), e)\}$. And a new structure $I_5 = I[\{\neg s(i(b)), \neg s(e), \neg p(b, i(b), e)\}]$ is constructed. The semantic tree is shown is Fig. 3.8.

5. In the partial hyper-linking of the last round, $C4$ is hyper-linked with eligible literal $\neg p(b, i(b), e)$ to generate ground instance $C13 = C4\alpha_4$, where $\alpha_4 = \{x \leftarrow b\}$.

$$C13 : \{p(b, i(b), e)\}$$

After $C13$ is added, the ground set $\{C8, C9, C10, C11, C12, C13\}$ is unsatisfiable and thus a proof has been found based on Herbrand theorem. The closed semantic tree in Fig. 3.9 shows the contradiction.

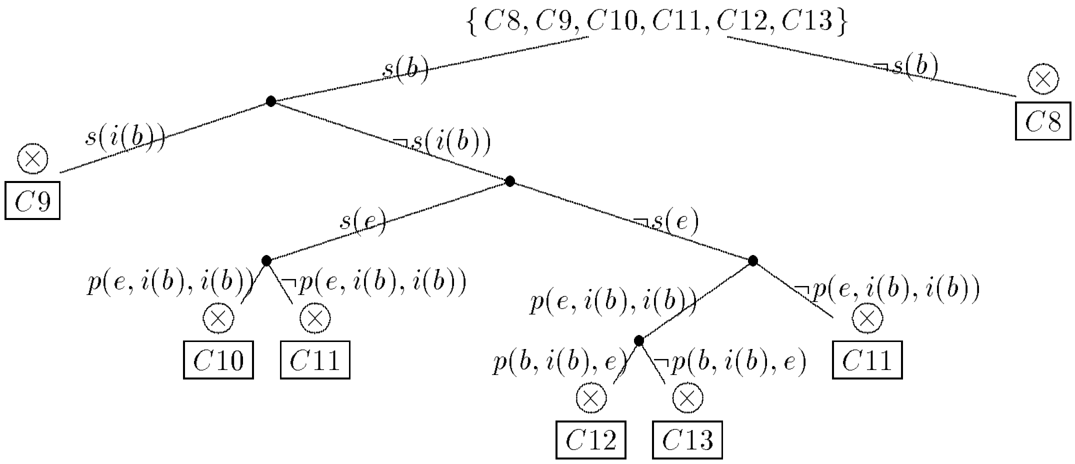


Figure 3.9: Proof found in round 5

Chapter 4

Model Finding

Semantic hyper-linking uses user-provided semantics to prune the search space but semantic checks are often expensive. Therefore, we want to check that, without using semantics, the models found for the ground clause set cause no direct or indirect contradictions (to be defined in the next section) with the non-ground input clauses. In other words, for a given structure I_i determined by the input structure I and a model M_i (of which $\mathcal{L}_i$ is a subset), we want to derive, using syntactic inference rules, a logical consequence D of the input clauses such that $I_i \not\models D$. If such D can be found, we consider M_i to be "bad" because a ground clause (D) has been found to contradict M_i. So there is no need to carry out a round of semantic hyper-linking with I and M_i, and we can backtrack to find another model for the ground clause set. Although this problem is undecidable in general—it is basically first-order theorem proving, we have devised several procedures to partially achieve this and allow us to choose "better" models before the "bad" ones.

In this chapter, we discuss the model finding process, called *incremental model finding*, and several searching strategies used in semantic hyper-linking to search for better models for the ground clause set. In the next section, we will discuss the incremental model finding strategies in detail. Then we will discuss the completeness of those strategies and one enhancement to the methods. We also present results from some hard theorems to show that those model finding strategies help to find proofs faster.

4.1 Introduction

Semantic hyper-linking conducts a model searching process to search for models M_2, M_3, ..., M_n, ... for ground clause sets $\{ D_1 \}$, $\{ D_1, D_2 \}$, ..., $\{ D_1, \ldots, D_{n-1} \}$, ... respectively, and obtain structures I_2, I_3, ..., I_n, ... Such model searching can be seen as an exploration of a closed semantic tree for the input clause set. Each D_i corresponds to a contradiction node that closes the path containing M_i.

In such a closed semantic tree exploration, we are concerned about efficiency in two aspects: First, during the model searching, redundancies should be avoided whenever possible—closed branches should not be searched again. Secondly, the exploration of the semantic tree is to be kept as small as possible, namely, the contradiction nodes should have depth as small as possible in the semantic tree. This allows us to have a smaller search space. The model finding strategies we use with semantic hyper-linking address these two concerns to gain efficiency.

To avoid the redundancies in search of a model for the ground clause set $\{ D_1, D_2, \ldots, D_{n-1} \}$, we use a method called *incremental model finding*: The old model M_n is kept and we search for a new model M_{n+1}, after D_n is generated, based on the old one. This is based on the following simple observation.

Proposition 59. *Given a structure I for a clause set S, if $I \not\models S$, then for any superset S' of S, $I \not\models S'$.*

Proof. $I \not\models S$ if any only if there exists a clause C in S such that $I \not\models C$. Since C also appears in any superset S' of S, it easily follows that $I \not\models S'$. □

The idea of incremental model finding is this: We continue the PC prover from the node in the semantic tree where M_{n-1} was found the last time. New models are constructed *incrementally* in such a way that closed branches of the semantic tree are never visited again.

The model finding is called incremental because new ground clauses might gradually expand the exploration of a closed semantic tree. Such expansions result from the introduction of new literals in $C_2\alpha\beta$ (see Section 3.3) of which corresponding atoms do not occur in the ground clause set. Introduction of such new literals enlarges the search space (in the closed semantic tree exploration) and often makes it harder to find the proof. One way to avoid introducing new literals in the semantic tree exploration is to find instances or logical consequences of input clauses which contradict the model.

Definition 60. A model M of ground literals *directly contradicts* a clause C if and only if C has a ground instance $C\theta$ such that for each literal L in $C\theta$ there is a literal $\neg L$ in M.

Definition 61. Given a model M for a set of ground instances of clauses in a clause set S, M *directly contradicts* S if and only if M directly contradicts a clause in S. It is decidable to check direct contradiction if S and M are finite. M *indirectly contradicts* S if and only if M directly contradicts a logical consequence of S. In general it is undecidable to check indirect contradictions. $\square$

Proposition 62. *For a clause set S, let C be a logical consequence of S, and I be a structure for S. If $I \not\models C$, then $I \not\models S$.*

Proof. It is known that $S \models C$ and thus $\neg C \models \neg S$. Also, if $I \not\models C$ then $I \models \neg C$. So $I \models \neg S$, or equivalently, $I \not\models S$. $\square$

Since the model M_i for the set of ground instances of clauses in S is subset of the related structure I_i, if M_i is in direct or indirect contradiction to S, it implies that I_i is also in contradiction to S. And the model finding process should backtrack to find a new model (and thus a new structure). Therefore it is useful to identify direct or indirect contradiction between a model M_i and the input clauses before M_i is used in semantic hyper-linking to generate more clauses and introduce new literals.

The approach used to detect contradiction between M_i and the input clause set S, is based on the following theorem.

Theorem 63. *Let S be a set of clauses and M be a model for a set of ground instances of clauses in S. Let S_M be the set of unit clauses such that $\{ L \} \in S_M$ if and only if $M \models L$. $S_M \cup S$ is unsatisfiable if and only if $M \not\models S$.*

Proof.

	$S_M \cup S$ is unsatisfiable
if and only if	every structure that is a model for S_M, is not a model for S
if and only if	there is no structure that is a model for both M and S
if and only if	$M \not\models S$.

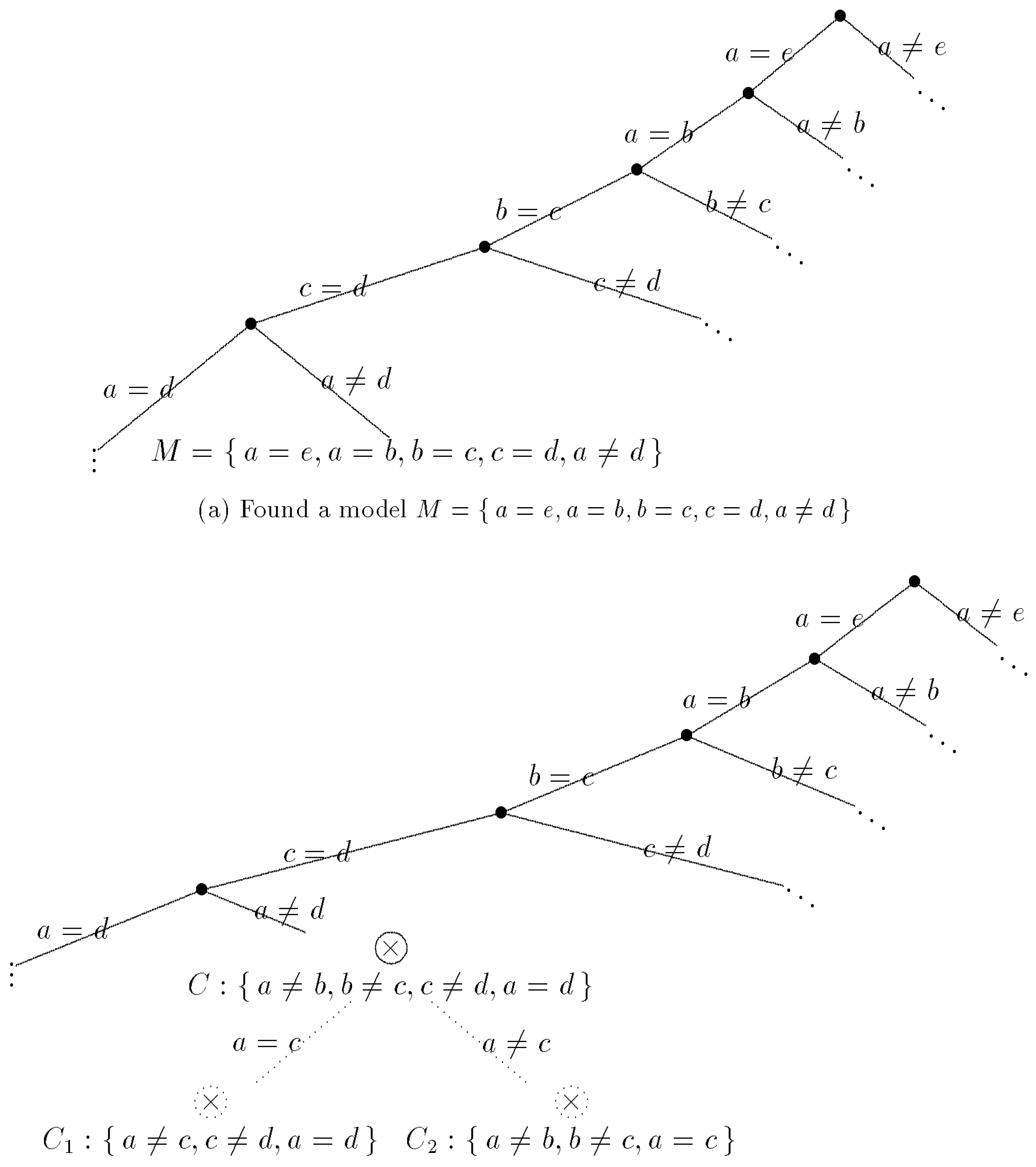
□

In practice, we begin the contradiction check with the input clause set S plus the set S_M of ground units obtained from the model M_i. Clearly this is a first-order theorem proving problem, but we only want to check, with a reasonable amount of effort, that $S_M \cup S$ is unsatisfiable. If a contradiction is detected, we generate a ground clause (an instance or a logical consequence of the input clauses) in direct contradiction to M_i and model finding needs to backtrack to find a new model; otherwise, M_i is used in the following round of semantic hyper-linking.

Example 19. In Example 18 (page 55), given a model $M = \{ a = e, a = b, b = c, c = d, a \neq d \}$ and the transitivity axiom $\{x \neq y, y \neq z, x = z\}$, $a = d$ is derived to implicitly contradict the model M. Thus we can close the path corresponding to M in the semantic tree by a contradicted clause $\{ a \neq b, b \neq c, c \neq d, a = d \}$ which is a logical consequence obtained by resolving two ground instances, $\{ a \neq b, b \neq c, a = c \}$ and $\{ a \neq c, c \neq d, a = d \}$, of the transitivity axiom upon literals $a = c$ and $a \neq c$.

Figure 4.1 illustrates the idea. In Fig. 4.1(a), incremental model finding found M. Fig. 4.1(b) shows that neither $M \cup \{ a = c \}$ nor $M \cup \{ a \neq c \}$ satisfies the transitivity axiom. Such a contradiction is found by a logical consequence C without generating $a = c$ or $a \neq c$ in the ground clause set. C not only closes the path but also avoids any model containing subset $\{ a = b, b = c, c = d, a \neq d \}$. Note that cases of $a = c$ and $a \neq c$ are closed by instances of the transitivity axiom. If C were not generated, exploration of the semantic tree might go "deeper" and introduce more literals (for example, $a = c$ or $a \neq c$) which enlarge the search space. Note that C is a resolvent from C_1 and C_2 by resolving upon $a = c$ and $a \neq c$. □

During the search for a model of the ground clause set, four different methods are applied to detect and discard models causing contradictions: *model filtering*, *model literal replacement*, *UR resolution* and *simple term rewriting*. Those methods try to check, without expanding the semantic tree with new atoms, if a model contradicts



(a) Found a model $M = \{a = e, a = b, b = c, c = d, a \neq d\}$

(b) Close the path by clause C without introducing $a = c$

Figure 4.1: General idea of incremental model finding

the input clauses. *Contradicted ground instances*, which are in direct contradiction of the model and are either instances or logical consequences of input clauses, are generated and kept to avoid the same contradiction in the new models.

4.2 Incremental Model Finding

The incremental model finding method is basically a modified PC prover. One major difference is that it starts with an old model found earlier and finds a new model based on the old one; the PC prover in Lee's prover [Lee90, LP92] does not use old models and stops as soon as the clause set is found to be satisfiable, often before a model is found. Thus we maintain only *one* semantic tree (possibly infinite) throughout the search for a proof. In each incremental model finding, backtracking takes place on failure nodes (which contain **F**); subtrees of failure nodes are discarded; closed branches (terminated with failure nodes) are never visited again; and the semantic tree is expanded when necessary.

Like the PC prover, model finding can be thought of as a preorder traversal of the semantic tree starting from the root. The two sub-cases of each node represent the case analysis on one literal: one is true in the user-provided semantics, the other is false. To search for a new model when new ground clauses are generated, we continue from the node where the previous model is found. Additionally, in case analysis the true case is always chosen first. This is a heuristic method to reduce the number of eligible literals in the model: The first cases correspond to non-eligible literals and the second cases often need not be carried out as a result of intelligent pruning of the search space.

4.3 Model Filtering

After a model M is found for the ground clause set, M might directly contradict the input clauses, namely, there might be ground instances (not yet generated) of the input clauses which falsify the model M. We use *model filtering* to "filter out" such models and make sure that the model we found does not directly contradict the input clause set. From the viewpoint of semantic tree exploration, model filtering can be seen as identifying contradiction nodes through direct contradiction with input clauses.

The filtering idea is simple: Given a model $M = \{ L_1, L_2, \ldots, L_n \}$, filtering checks if there is a ground instance $C\theta$ of the input clause C such that for each literal L in $C\theta$, $L = \neg L_i$ for some i. One simple way is to hyper-link input clauses with the literals in the model. If input clause C is fully hyper-linked with the model literals to get $C\theta$ where θ is a ground substitution (because all literals in the model are ground), the model M immediately contradicts $C\theta$. This is because for every literal $L\theta$ in $C\theta$ there is a complementary literal L_i in the model. We then add $C\theta$, which is a ground instance of an input clause C, to the ground clause set to avoid generating the models causing the same contradiction. Note that model filtering is essentially the same as a subsumption test to check if there is any C that subsumes $\neg M$ and if so, C contradicts M.

Example 20. Suppose we obtain a model $\{ p(a), q(b), \neg r(a,b), \ldots \}$. Suppose there exists an input clause $\{ \neg p(x), \neg q(y), r(x,y) \}$. The model contradicts this clause and thus should not be kept; the model finding should backtrack to find another model. The ground clause $\{ \neg p(a), \neg q(b), r(a,b) \}$ is added to the ground clause set to avoid generating any other model containing the subset $\{ p(a), q(b), \neg r(a,b) \}$.

As another example, consider Fig. 4.1(b). If the case analysis on literal $a = c$ is carried out, model filtering can detect the direct contradictions and generate C_1 and C_2 respectively. □

If more than one input clause is contradicted by the model, we only add the smallest ground instance of those to the ground clause set. In general it is not necessary to add all contradicted instances because the model will be changed and probably new models remove all other contradictions. Another reason is that model finding, which is basically a modified PC prover, is fast and contradicted ground instances will be added eventually if they still cause contradictions in the new models.

In fact, those contradicted ground instances could still possibly be found in later rounds of semantic hyper-linking if model filtering is not used. However, model filtering detects the contradictions as early as possible to save some rounds of semantic hyper-linking. Also model filtering detects such contradictions without expanding the semantic tree and thus helps to find the proofs sooner.

Manthey and Bry [MB88] used the same technique to check a model in their model-generation theorem prover SATCHMO. They used a different way to generate ground literals (by utilizing the range-restricted properties—that every variable occurs in at least one negative literal—in a forward-chaining fashion), which constitute a model for the input clauses. In SATCHMO the instances of the contradicted input clauses are not saved; the contradiction only causes backtracking in the search for a proof.

Obviously model filtering is sound since it generates ground instances of input clauses.

4.4 Model Literal Replacement

Lee and Plaisted [Lee90, LP92] used *predicate replacement* to generate instances of input clauses faster by partially hyper-linking literals in a clause. *Replacement rules* are constructed manually from input clauses to apply predicate replacement.

Definition 64. A replacement rule is a clause

$$\{ \neg M_1, \neg M_2, \ldots, \neg M_n, L_1, L_2, \ldots, L_m \}$$

in the format of

$$M_1, M_2, \ldots, M_n \rightarrow L_1, L_2, \ldots, L_m$$

$M_i, 1 \leq i \leq n$, and $L_j, 1 \leq j \leq m$ could be positive or negative literals. Literals $\neg M_1, \neg M_2, \ldots, \neg M_n$, to the left of the arrow are called *distinguished literals*. □

In predicate replacement, only distinguished literals in a replacement rule are hyper-linked with literals from other clauses. This is essentially a "partial" hyper-linking technique, and provides a faster way to generate instances using hyper-linking—some instances might not be generated if full hyper-linking is required. In each round of hyper-linking, users can apply predicate replacement repeatedly or just once to generate new instances. Predicate replacement has been used to efficiently prove many hard theorems, especially those in areas in which defined concepts are common (see [Lee90]). This success indicates that predicate replacement has a lot of potential in general theorem proving. However, predicate replacement is not complete and it often requires deep understanding of the theorems to devise proper replacement rules. For a novice user, it is difficult to take advantage of predicate replacement. Therefore, the replacement rules are not natural and sometimes it is not easy to devise proper replacement rules to obtain fast proofs.

We use a simplified predicate replacement called *model literal replacement*—we only apply the replacement using the model literals, treated as unit clauses, as well as the replacement rules generated from the input clauses. As in predicate replacement, the goal of model literal replacement is to derive a contradiction through replacement involving model literals, namely, to check if a model is in indirect contradiction to the input clauses. There are two types of replacement rules used in the model literal replacement—*natural replacement rules* and *minimal replacement rules*. Although each type of replacement rules correspond to a different philosophy, both can be generated automatically from the input clauses. Thus human intervention is not needed.

4.4.1 Natural Replacement

Natural replacement rules are defined as follows:

Definition 65. A replacement rule is *natural* if

(1) it has a unit consequence, namely, it has the format

$$M_1, M_2, \ldots, M_n \longrightarrow L$$

where L and each M_i are literals.

(2) each variable in L occurs in some M_i. □

Example 21. The axiom $\{ x = y, y = x \}$ has two replacement rules: $x \neq y \rightarrow y = x$ and $y \neq x \rightarrow x = y$. The axiom $\{ i(x,u) \neq i(x,v), u = v \}$ has only one replacement rule: $i(x,u) = i(x,v) \rightarrow u = v$. □

With the first condition and every M_i being unified with a literal from a unit clause, we obtain a unit clause $\{ L\theta \}$ because all M_i are (unit) deleted. In this case, we can think of the replacement as deriving a unit consequence $L\theta$ from single facts $M_i\theta$. The second condition guarantees that once all M_i in a natural replacement rule are unified with ground literals, L becomes ground after a proper substitution is applied. We use input clauses to generate all possible natural replacement rules. Natural replacement rules are then applied with model literals in *natural replacement.*

Here is how natural replacement works: We think of the literals in the model as literals from (ground) unit clauses (called *unit literals*) which can be seen as unit facts. We then use the natural replacement rules to generate more unit literals which can be seen as unit consequences or derived unit facts. New unit literals are checked to see if there is any contradiction with old ones. We can repeat the replacement more than once, just like the repeated replacement rules used in [Lee90, LP92]. However, repeated natural replacement may not terminate. For example, the natural replacement rule $p(x) \longrightarrow p(f(x))$ might keep generating larger and larger $p(f(f(\ldots)))$ literals and will not terminate. Thus a bound on the number of replacements or the number of the derived unit literals, is needed to guarantee termination; bounding the time spent in replacement is another choice. The natural replacement procedure is described in Fig. 4.2.

Once a contradiction is detected, a contradicted ground instance can be generated as follows: For each derived literal L, we maintain a set of model literals which generate L using natural replacement. A contradiction is found if a newly derived literal L contradicts an existing literal $\neg L$. Suppose that L depends on the set S_L of model literals; $\neg L$ depends on the set S_{NL}. Then we add a clause $C = \{ M : \neg M \in S \}$ to the ground clause set, where $S = S_L \cup S_{NL}$. It is sound to add C as a ground clause because C is a logical consequence of the input clauses. To see why C is a logical consequence of the input clauses, we can think of C as obtained by a sequence of binary resolutions by resolving upon non-model literals from the ground instances, starting with the ones containing L and $\neg L$, involved in the related natural replacement steps.

Any further models in contradiction to C would also indirectly contradict the input clauses. Thus C can be added to the ground clause set, and we not only change the model but also avoid generating any model later that would cause the same contradiction.

Example 22. Let us look at a simple example in the ground case. Suppose model literals L_1 and L_2 generate L using the natural replacement rule $L_1, L_2 \longrightarrow L$, and L_3 and L_4 generate $\neg L$ using the natural replacement rule $L_3, L_4 \longrightarrow \neg L$. Then L

depends on the model literals $\{ L_1, L_2 \} = S_L$, and $\neg L$ depends on $\{ L_3, L_4 \} = S_{NL}$. Since L and $\neg L$ cause a contradiction, the algorithm returns a set $S = S_L \cup S_{NL} = \{ L_1, L_2, L_3, L_4 \}$. We then add a clause $C = \{ \neg L_1, \neg L_2, \neg L_3, \neg L_4 \}$ to the ground clause set. To illustrate (informally) that C is a logical consequence of the input clauses, we can think of C as a resolvent from the two replacement rule instances by resolving on L and $\neg L$. Since the two replacement rules are actually input clauses, it easily follows that C is a logical consequence of the input clauses. C is then added to the ground clause set to avoid generating any model that contains the subset $\{ L_1, L_2, L_3, L_4 \}$. □

Example 23. Let us consider Example 19 once again to see how to derive the contradicted instance C. From the transitivity axiom $\{x \neq y, y \neq z, x = z\}$, we could generate three replacement rules, one of which is $R : x = y, y = z \rightarrow x = z$. With model literals $a = b$ and $b = c$, natural replacement applies R to obtain the new literal $a = c$. With the derived literal $a = c$ and model literal $c = d$, natural replacement generates another literal $a = d$ which contradicts model literal $a \neq d$. Model literal $a = b$, $b = c$, $c = d$ and $a \neq d$ are used to derive the contradiction. Thus the contradicted ground instance C is $\{ a \neq b, b \neq c, c \neq d, a = d \}$. □

4.4.2 Minimal Replacement

The second type of replacement rules are *minimal replacement rules.* The idea is an extension of natural replacement to allow more than one literal on the right-hand side of the rules, while at the same time to minimize (hence the term "minimal") the number of literals on the left-hand side.

Definition 66. A *minimal replacement rule* has the format

$$M_1, M_2, \ldots, M_n \rightarrow L_1, L_2, \ldots, L_m$$

such that

(1) every variable occurring in the replacement rule, appears in some M_i, $1 \leq i \leq n$.

(2) No subset of $\{ M_1, M_2, \ldots, M_n \}$ exists such that condition (1) still holds. □

Note that there could be more than one minimal replacement rule for an input clause.

Example 24. Consider the clause $\{\, el(z,x), el(z,y), \neg el(z, union(x,y)) \,\}$ for the set axiom

$$z \in x \cup y \supset z \in x \vee z \in y$$

Two minimal replacement rules can be generated from this clause:

$$el(z, union(x,y)) \rightarrow el(z,x), el(z,y)$$

and

$$\neg el(z,x), \neg el(z,y) \rightarrow \neg el(z, union(x,y))$$

The former rule naturally corresponds to the original axiom. □

Often minimal replacement rules corresponds to the defined concepts in the axioms, especially in such areas as set theory (as shown in the above example) and temporal logic.

Minimal replacement is basically a hyper-linking procedure to generate ground clauses by predicate replacement: Initially all literals in the model M are treated as units, then the normal hyper-linking procedure repeatedly derives more ground clauses by applying minimal replacement rules. Note that model literals are treated as units and literals hyper-linked with model literals are unit deleted. Since the goal of minimal replacement is to derive contradiction, a PC prover is applied after each round of this "small" hyper-linking procedure, and, as in natural replacement, each derived clause C is accompanied by the set of corresponding model literals used to derive C.

When a fully-linked unsatisfiable ground clause set U is obtained by minimal replacement, an indirect contradiction between the model M and the input clause set has been found. Thus, as in the natural replacement, a contradicted clause will be generated which is $\{\, L : \neg L \in$ the accompanying set of model literals for $C, C \in U \,\}$. Such contradicted clauses guarantee that the same indirect contradictions do not exist in new models.

The minimal replacement is sound and the soundness is guaranteed by the fact that each contradicted clause generated is a logical consequence of the input clauses.

Theorem 67. *If a contradiction is detected by minimal replacement, the contradicted instance is a logical consequence of the input clauses.*

Proof. It is known that resolvents are logical consequences of parent clauses. Soundness of minimal replacement can be proved by showing that, with resolution, the contradicted instance is a logical consequence of some ground instances of the input clauses.

Take the fully-linked unsatisfiable ground clause set U obtained by minimal replacement, and let U' be the new set from U such that literals hyper-linked with model literals, are not deleted. U' now contains ground instances of the input clauses. Then the contradicted clause can be obtained from U' using binary resolution by resolving upon literals originally not in the model M. $\square$

Example 25. Let us consider a simple example. Suppose we obtain a ground model $M = \{ L_1, L_2, L_3, L_4, \ldots \}$. Suppose the following minimal replacement rules are from the input clause set (we here only list the ground instances of the rules that would hyper-link the model literals).

$$\begin{array}{rcrcl} C_1 & : & L_1, L_2 & \rightarrow & L_5, L_6 \\ C_2 & : & L_3 & \rightarrow & \neg L_5 \\ C_3 & : & L_4 & \rightarrow & \neg L_6 \end{array}$$

Minimal replacement on M would start with unit clauses $\{ L_1 \}$, $\{ L_2 \}$, $\{ L_3 \}$, $\{ L_4 \}$, ... and generates ground clauses (among others)

$$\begin{array}{ll} \{ L_5, L_6 \} & \text{derived from } C_1 \text{ using } L_1 \text{ and } L_2 \\ \{ \neg L_5 \} & \text{derived from } C_2 \text{ using } L_3 \\ \{ \neg L_6 \} & \text{derived from } C_3 \text{ using } L_4 \end{array}$$

It can be easily shown that the first three clauses constitute a contradiction, which means M is in indirect contradiction with the input clauses. Then the contradicted clause would be $\{\neg L_1, \neg L_2, \neg L_3, \neg L_4\}$ which can be obtained by first applying resolution to C_1 and C_2 (resolving upon L_5 and $\neg L_5$) to obtain C', and then applying resolution to C_3 and C' (resolving upon L_6 and $\neg L_6$). □

4.4.3 Enhancements

In this section we describe several refinements of natural or minimal replacement. Experiments show that those refinements help to find proofs faster.

Model filtering after natural and minimal replacement

Those literals generated at the end of natural replacement can be seen as in a model for all the generated ground instances. Thus model filtering can be readily applied here which completes the contradiction checks for the last round without applying natural replacement and generating new literals. Once a contradiction is found this way, a contradicted ground instance can be generated to cause the model finding to backtrack.

For minimal replacement, a model is obtained for the generated ground clause set if satisfiable. And again model filtering can be applied here. Note that, however, if a contradiction is found by applying model filtering this way, the contradicted ground instances are generated to backtrack the model finding within minimal replacement, not the incremental model finding before minimal replacement. Only after an unsatisfiable ground set is obtained does minimal replacement generate a contradicted ground instance to backtrack the incremental model finding.

Using units from UR resolution

Unit clauses from UR resolution (discussed in the next section) can be used in model filtering as well as natural and minimal replacement. Such unit clauses are logic consequences of input clauses. Thus we can use them to help find indirect contradictions between the ground model and the input clauses. Especially, for those unit clauses that are large and will not be used until much later in the proof searching, due to the selecting scheme in UR resolution, sometimes they can be used to help find the proofs earlier.

Bounded minimal replacement

If no indirection contradiction is found by minimal replacement, we can continue minimal replacement in a restricted manner such that none of the generated clauses introduces new subterms; the only rules used are those that produce literals containing existing subterms. We call this *bounded minimal replacement*. Intuitively, this allows us to complete the unsatisfiable ground instance set if all necessary subterms are generated but some important literals are not yet generated.

4.4.4 Comparison with Predicate Replacement

In general both model literal replacements are basically hyper-linking procedures applying predicate replacement with model literals, treated as units, as initial input. The major differences between predicate replacement and the two model literal replacements are:

- Model replacement rules are generated automatically.
- Model literal replacements begin with unit clauses from model literals.
- A contradiction detected by model literal replacement is not a proof (as in predicate replacement) because the model literals are not really unit clauses; instead a contradiction indicates that the model is in indirect contradiction to the input clauses. Thus, a contradicted clause has to be generated to backtrack the incremental model finding and avoid the same indirect contradiction in any model found later.
- Generated ground clauses during replacement are not saved (and thus will not increase the search space of model finding).

4.5 UR Resolution

4.5.1 The Use of UR Resolvents

Unit resolution [WCR64] is a resolution involving one unit clause $U = \{L\}$ and another clause C (which could also be a unit clause). Suppose C has a literal M

and there exists a most general unifier θ such that $L\theta = \neg M\theta$. After applying unit resolution on U and C, the resolvent is $(C - M)\theta$. Since a unit clause has no other literals than the one resolved upon, the resolvent from unit resolution does not introduce new literals. This is compatible with the philosophy of hyper-linking that literals from different clauses should not be combined.

UR (*U*nit *R*esulting) resolution [WOLB84] is a sequence of unit resolutions in which the resolvent (called *UR resolvent*) is a unit clause. This can be seen as a multi-step unit resolution and is particularly helpful because the UR resolvents can then be used in further UR resolution. UR resolution has been shown to be a useful theorem proving technique. In particular, unlike general binary resolution, UR resolution does not combine literals from different clauses. This is compatible with the philosophy of instance-based theorem proving and fits in with our methods.

UR resolvents can be used to detect unit conflicts during model finding or after a model is found (in which case backtracking would take place).

We use UR resolution to detect a contradiction caused by the model found during model finding. The basic idea is to check if derived UR resolvents contradict any literal in the model. To achieve this, the model finding procedure is modified so that a case analysis on literal L also checks if L contradicts any unit clause $\{ R \}$, namely, $R\theta = \neg L$ for some unifier θ. This is a special case of *unit conflict check* and only needs fast one-way matching instead of two-way unification.

With a ground model, derived UR resolvents are also used in unit conflict checks with the model (in the way similar to model filtering). Model backtracking takes place if such a contradiction is found. With the unit clause, no other model containing L can be generated (since there are unit conflict checks during model finding).

4.5.2 Comparison with Natural Replacement

UR resolution and natural replacement seem quite similar. The major difference is that we might save intermediate UR resolvents since they are logical consequences of input clauses. In natural predicate replacement we do not save any intermediate results because literals in the model are not really from unit clauses.

Like natural replacement, UR resolution might not terminate. So a time bound or a bound on the number of derived resolvents needs to be used. The bound also provides a way to balance the work by UR resolution and other proving techniques in our prover. When resuming its execution, UR resolution continues from where it was stopped last time.

4.5.3 Selection of UR Resolvents

Derived UR resolvents can be used in further UR resolution. Since unit conflict during UR resolution signifies a proof being found, selection of derived UR resolvents for the next UR resolution often has profound impact on how fast such a UR proof can be found.

One way to use a selected derived UR resolvent U is to assure that all possible UR resolutions involving U at least once with possibly other "old" units, are carried out, and after which U is considered "old." This can avoid duplications and allows a systematic development of UR resolution.

Here we discuss two criteria to select a "smallest" new UR resolvent U: One is based on the total size (the number of symbols appearing in U) and the other on the proof size (the number of steps of UR resolution to derive U). The philosophy is to prefer units that are generated earlier (thus with smaller proof sizes) or have smaller total sizes.

In implementation, there are two types of UR resolutions for each criterion. UR resolution based on the total size is commonly used in other theorem provers and thus we emphasize its role over the one based on the proof size. Units generated in UR resolution based on the total size are also used, with zero proof size indicating their importance, in UR based on the proof size. This provides some "cooperation" between these two kinds of UR resolutions. From the experimental results, we have found that some UR proofs are found by UR resolution based on the proof size even though large units are involved and the aforementioned cooperation seems to help.

4.6 Term Rewriting

Term rewriting [Plaar, DJ90] is an important and powerful technique in equational theorem proving in which equations play an important role. The main idea is to use a rewrite rule $r \rightarrow s$, which comes from an equation $r = s$ (or $s = r$), to replace any instance of r by s, namely, replacing equals by equals. Term rewriting can be applied at any term level. For example, with rewrite rule $a \rightarrow b$, literal $p(a, c)$ can be rewritten to $p(b, c)$. The rewrite rules are usually chosen by orienting the equations to carry out only one way rewriting (to avoid circular rewriting) and, more importantly, ensure termination. Choosing an ordering on terms to orient the equations into rewrite rules and ensure termination, has been one major issue in term rewriting research [Der87].

Rewriting only at the top level, namely at the predicate level, has also been studied. With such mechanisms, predicates can be replaced by their definitions, like human mathematicians usually do. Plaisted and Potter [PP91] use a replacement rule of the form $replace(exp_1, exp_2)$ in the theorem prover SPRFN [Pla82, Pla88] based on a natural deduction system in first order logic [Pla82, Pla88]. The replacement rule signifies that all subgoals of the form exp_1 should be replaced by subgoals of the form exp_2. The replacement rule is sound if $exp_2 \supset exp_1$. Predicate replacement by Lee and Plaisted and the model literal replacement discussed in Section 4.4, are all based on this idea.

When semantic hyper-linking tries to prove problems involving equations (not necessarily in unit clauses), the ground models found by incremental model finding usually have ground equations. Also, UR resolution will generate equations. Such equations suggest that term rewriting can be applied to detect indirect contradictions between the ground models and the input clauses (although the input clauses might not have equations as unit clauses).

We employ a simple term rewriting method on the model literals using the equations from the ground models and units generated by UR resolution. Detecting an indirect contradiction this way is equivalent to detecting a contradiction by applying term rewriting on model literals (seen as unit clauses) using the equations from the ground models and unit clauses.

To orient an equation and obtain a rewrite rule, one must first define an ordering among terms. In particular, we are interested in orderings that guarantee the rewriting terminates.

Definition 68 [Plaar]. A *termination ordering* on terms is a partial ordering $\prec$ which is well-founded and satisfies the *full invariance property*, that is, if $s \prec t$ and θ is a substitution then $s\theta \prec t\theta$, and also satisfies the *replacement property*, that is, $s \prec t$ implies $f(\ldots s \ldots) \prec f(\ldots t \ldots)$.

Term rewriting used in our prover is simple because it adopts a simple term ordering that is not a termination ordering and might not do some rewriting that could have been done using other termination orderings (such as lpo, the lexicographic path ordering [Plaar, Der87]). Since equations are mainly from model literals and model literals are those to be rewritten most of the time, we are mainly interested in ordering of ground terms but also allow non-ground equations to be oriented to rewrite rules and be rewritten in a restricted way.

Let $|s|$ for a ground term s be the number of symbol (constants and functions) occurrences of s. We define the ordering $\prec$ on ground terms as follows: For two ground terms s and t, $s \prec t$ if $|s| < |t|$ or, when $|s| = |t|$, s is less than t lexicographically when written as strings. Ground equations are oriented following this ordering such that $s \rightarrow t$ are generated with $s \prec t$.

Non-ground equations are oriented based on this simple idea: $s \prec t$ if $|s| < |t|$ and every variable in s occurs in t at least as many times as in s. Also, $s \prec t$ if $|s| = |t|$ and every variable in s appears in t at least as many times as in s, and the subterms are ordered lexicographically. That is, $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots t_n)$ and there exist one i such that $s_i \prec t_i$ and for all $j < i$, $s_j = t_j$. Another method to use non-ground equations is not to orient the equations, namely use both $s \leftarrow t$ and $t \leftarrow s$ for an equation $s = t$.

Ground literals are rewritten according to the lexicographic ordering. To rewrite a non-ground literal L, we ground it to get L'. We ground it by introducing new constant symbols $c_1, \ldots c_n$ with $c_1 \prec c_2 \prec \ldots \prec c_n$ and replacing the variables (left to right) by $c_1, \ldots c_n$. Then we rewrite L' using ground rewriting to get M'. Then in M' we replace $c_1, \ldots c_n$ back by the original variables to get M. Then we say L rewrites to M.

Simple term rewriting terminates since each rewriting of a ground term reduces it in lexicographic ordering which is well-founded. Also, rewriting non-ground terms is done in ground case and the "un-grounding" process reduces the original non-ground term in lexicographic ordering. So the terms keep decreasing and the process must terminate.

A contradiction occurs if either (1) there are two rewritten literals L_1 and L_2

such that there exists a mgu θ and $L_1\theta = \neg L_2\theta$, or (2) a rewritten literal $s = s$ is generated. As in other model finding strategies detecting model contradictions, model literals deriving each rewritten literals are kept and a contradicted instance can be constructed once a model contradiction is found. It can also be seen that detecting model contradictions using this simple term rewriting is also sound.

4.7 Soundness and Completeness

Semantic hyper-linking is complete and sound (as shown in Section 3.5). In this chapter we introduce several model finding strategies— model filtering, model literal replacement, UR resolution, and term rewriting—with semantic hyper-linking to find proofs sooner. One major concern about the model searching strategies is whether they still maintain the completeness and soundness of semantic hyper-linking.

4.7.1 Soundness

As we have shown in previous sections, those model finding strategies all generate ground logical consequences of the input clauses, so the soundness of semantic hyper-linking is preserved.

4.7.2 Completeness

In this section, we will show completeness is still preserved by those model finding strategies; in other words, if an input clause set S is unsatisfiable, semantic hyper-linking augmented with the model finding strategies can still find a proof.

In the proof process using semantic hyper-linking, a sequence of $I_1, I_2, \ldots$ of structures are chosen and a sequence of $D_1, D_2, \ldots$ of ground instances are chosen such that

$$\text{for all } i, I_i \not\models D_i; \text{ and for all } i < j, I_j \models D_i$$

Each I_i is constructed by extending a ground model M_i for $D_1, \ldots, D_{i-1}$. The completeness proof of semantic hyper-linking is independent of how the ground model M_i (and thus a new structure I_i) is chosen. However, it should be guaranteed that a minimal complexity D_i be chosen (in some pre-defined complexity ordering) such that $I_i \not\models D_i$. *Any* model M_i can be chosen for $D_1, D_2, \ldots, D_{i-1}$. For a ground set, there might be more than one such ground model. Model finding strategies help to filter out the "bad" choices, namely, those ground models found to be in indirect contradiction with the input clauses.

Theorem 69. *When the model finding strategies are used to find the models M_i for ground instances $D_1, \ldots, D_{i-1}$, semantic hyper-linking is complete.*

Proof. The completeness proof of the model finding strategies is based on the following observations:

1. Given a ground clause set G, those model finding strategies only generate clauses containing atoms, or their negations, in G.

2. Those model finding strategies always terminate with a model M for $G \cup \{\text{contradicted instances generated}\}$, since there are finite number of possible contradicted instances to be generated (subject to a execution bound).

3. M is still a model of G (without the added ground clauses).

In other words, the model finding strategies can be thought of as some heuristics to finding a *better* model M_i for G (and thus a better interpretation I_i toward generating an unsatisfiable ground set). Since the completeness proof of semantic hyper-linking is independent of how M_i (or I_i) is chosen subject to the above conditions, completeness is retained. $\square$

4.8 Experimental Results

To illustrate the power of the model finding strategies, let us look at some results. Figure 4.1 shows how our prover performs on some problems when model finding strategies are used. The prover is implemented in Prolog and the results were obtained on a DEC5500.

IMV [BB82] (intermediate value theorem in real analysis) and AM8 [Ble83] (attaining maximum theorem in real analysis) are two very hard theorems; these two

Problem	With Model Finding					Without Model Finding	
	Clauses generated by			Time	Round	Time	Round
	Filtering	Replacement	UR	(sec)		(sec)	
IMV	8	2	0	357	4	2,172	10
AM8	0	1	0	2,586	2	> 10,000	—
EXQ1	0	2	0	96	2	302	6
EXQ2	13	8	0	106	1	> 2,000	—
EXQ3	9	11	0	90	2	> 2,000	—
HALT	2	2	0	125	1	> 2,000	—

Table 4.1: Model finding strategies help to find proofs fast

Round	Filtering	Replacement	UR
1	0	0	0
2	0	0	0
3	3 (3)	0	0
4	1 (1)	0	0
5	4 (3)	2 (2)	0
Total	8 (7)	2 (2)	0

Table 4.2: Ground clauses generated by model finding in IMV proof

hard theorems often generate very large search spaces that many powerful general theorem provers (for example, `Otter` [McC90] and CLIN [Lee90]) cannot handle. With proper semantics, semantic hyper-linking can prove IMV and AM8 reasonably fast[1]. If the model finding strategies are not used, AM8 runs over 10,000 seconds and still cannot find the proof; IMV takes 2,172 seconds and 10 rounds (it is faster because of UR resolution, which takes a long time, is disabled).

The proof of IMV is found by an unsatisfiable ground clause set containing 12 clauses. Figure 4.2 shows the numbers of clauses generated by each model finding strategy during the proof of IMV. Numbers in parentheses represent the clauses in the unsatisfiable set, namely the "useful" clauses for the proof. Out of 12 clauses in the unsatisfiable set, 9 of them are generated by the model finding strategies which generate in total 6 ground clauses.

EXQ problems, EXQ1–3, are three examples from quantification theory [Wan65] in the order of increasing difficulty. They are difficult for most powerful theorem

[1]In early versions of the prover in which some features were not yet implemented, IMV could be proved in 39 sec and AM8 in 492 sec.

provers probably due to the non-Horn property. We proved all three EXQ problems on the first try fairly quickly. We think EXQ3 has never been proven before fully automatically.

HALT is the undecidability of the halting problem [Bru91]. We use the version from non-standard clause translation [PG86]. Since it is difficult to devise reasonable semantics for the halting problem, we used a trivial semantics in which the true literals are exactly those positive ones. Even with such a bad choice for semantics, our prover still can prove it easily.

AND is the Andrew's challenge problem [Pel86] which could generate more than 1,600 clauses using standard clause translation. Again we used the version from non-standard clause translation. This version seems to be difficult for resolution provers like `Otter`.

Although not shown in Fig. 4.1, UR resolution find relevant clauses in some other problems. Also, UR resolution is a powerful proving technique of its own. It has helped to prove many other problems.

4.9 Summary

In each round of semantic hyper-linking, ground clauses are generated to "change" the existing model and search for a new model (thus a new structure), until the ground set is unsatisfiable. However, the search space of semantic hyper-linking grows when new literals are generated (for example, see Section 3.6). A larger search space implies that the proof is more difficult to obtain. In this chapter, we have described three model finding strategies that change the models without using semantics which is often expensive; at the same time, they do not increase the search space because they only generate clauses containing negations of model literals.

It is also interesting to note that, from the viewpoint of the semantic tree exploration, the ground clauses generated by the model finding strategies, correspond to contradiction nodes of which all the subtrees can still be closed if the model finding strategies are not used but that will require more searching and the proofs will be found later this way.

Those strategies used in incremental model finding still retain the completeness of semantic hyper-linking. They also maintain soundness since they all generate ground logical consequences of the input clauses. More importantly, they have a positive impact on the performance of semantic hyper-linking. In the proofs we have obtained so far, the model finding strategies almost always help.

```
Algorithm Natural Replacement(M, Replace, Bound, S)
input
   M: set of literals in the model
   Replace: set of natural replacement rules
   Bound: bound on the application of replacement
output
   S: set of literals in M, that derive contradiction
return values
   F: if no contradiction is found
   T: if a contradiction is found
begin
   % S_unit is a set of unit literals
   S_unit = M
   while Bound is not exceeded do
      for each natural replacement rule R = M_1, ..., M_n ⟶ L in Replace do
         for each Lθ obtained by unifying all M_i with literals in S_unit do
            if Lθ contradicts any literal in S_unit then
               % A contradiction is found
               S = { literals in M that derive Lθ or ¬Lθ }
               return T
            else
               S_unit = S_unit ∪ { Lθ }

   % No contradiction is found
   return F
end
```

Figure 4.2: Natural replacement algorithm

Chapter 5

Rough Resolution

5.1 Motivation

To ensure the completeness of semantic hyper-linking, a complexity ordering $\prec$ needs to be used and each D_i has to be chosen such that it is the smallest, with respect to $\prec$, ground instance to satisfy $I_i \not\models D_i$. For example, we can have an ordering $\prec_s$ on literals based on the number of symbols occurring in the literal. Such restriction on instance generation of semantic hyper-linking, however, imposes difficulties generating "large" ground instances of a literal since there might be many smaller (yet irrelevant) ones that need to be generated first.

As an example, consider the LIM+ problems [Ble90]. For such problems, given natural semantics, a correct ground instance of the goal clause

$$(D \leq 0) \vee \neg(|f(xs(D)) - f(a)| + |g(xs(D)) - g(a)| \leq e0)$$

has to be generated by substituting variable D with $min(d1(ha(e0)), d2(ha(e0)))$ to obtain the proof. Suppose $\prec_s$ is used. Then the ground instance is large and, with natural semantics, semantic hyper-linking generates many irrelevant smaller instances and diverts the search. Similar problems happen when large ground literals need to be generated in axiom clauses. In particular, semantic hyper-linking suffers from such smallest-first scheme most when large literals are needed to obtain the proof (like the correct ground instance of the goal of LIM+ is needed for semantic hyper-linking to find the proof).

One solution to this type of problem is using better semantics so smaller and irrelevant literals are not generated. But designing good semantics in general is difficult since very few, if any, semantics can capture all properties expressed by the axioms and, more importantly, the choices of decidable semantics to gain efficiency are not arbitrary (to be discussed in Chapter 6). Thus it is not practical to rely on better semantics to generate large literals before the search space grows out of control.

Another solution, which will be discussed in this chapter, is *not* to generate the large literals at all. We note that, during resolution, literals resolved upon are removed from the resolvents. If large literals are resolved upon, then they will be removed in the resolvents and instances of the resolvents, instead of the parent clauses, do not have large literals and can be used to obtain the proofs. Motivated by the above observations, we use *rough resolution* to address the problem by resolving upon (and deleting) literals that are "potentially" large (to be defined later). Thus large ground literals are not needed if ground instances of the resolvents are used, and proofs are easier to obtain by semantic hyper-linking. However, rough resolution is against the philosophy of hyper-linking, namely, not to combine literals from different clauses. We will discuss several restrictions on application of such resolution and reducing the number of retained resolvents so duplication of search space (by combining literals from different clauses) is not a serious problem as in ordinary resolution, and at the same time enjoying the benefits of removing large literals.

In this chapter we will discuss in detail the rough resolution technique. An example (LIM3 problem) is given to help illustrate the ideas. We will also give some test results to show that rough resolution is in general a positive refinement to semantic hyper-linking.

5.2 Rough Resolution

The basic idea of rough resolution is simple: we only resolve on those literals that are potentially large in ground instances. Those literals resolved upon are deleted in the resolvents and their ground instances need not be generated. If such large ground literals are used in the proof, rough resolution can relieve instance generation of generating such large literals and help semantic hyper-linking find the proof faster.

In the following discussions, we always assume that the complexity ordering on literals is $\prec_s$ (based on the number of symbols appearing in literals, not counting negation). For example, clause $C = \{\neg p(x, y, z), q(z, x)\}$ (where x, y and z are variables) has literals of size 4 and 3. Furthermore, $\neg p(x, y, z)$ is the largest literal in C, and is larger than $q(z, x)$ in any instance of C.

Example 26. Consider the following clauses (x, y and z are variables):

$$\begin{aligned} C_1 &= \{\, \neg p(x, y, f(z)), q(z, x) \,\} \\ C_2 &= \{\, p(f(x), g(y), z), \neg r(x, y) \,\} \end{aligned}$$

It is easy to see that the first literals of C_1 and C_2 are the largest ones in any instance.

Now suppose the following two ground instances are needed in the proof which needs no other instances of $p(f(a), g(h(b)), f(c))$:

$$\begin{aligned} C_1' &= \{\, \neg p(f(a), g(h(b)), f(c)), q(c, f(a)) \,\} \\ C_2' &= \{\, p(f(a), g(h(b)), f(c)), \neg r(a, h(b)) \,\} \end{aligned}$$

We can resolve C_1 and C_2 upon the first literals and get

$$C = \{\, q(z, f(x)), \neg r(x, y) \,\}$$

Then, instead of C_1' and C_2', a smaller ground instance

$$C' = \{\, q(c, f(a)), \neg r(a, h(b)) \,\}$$

can be used in the proof and avoids the use of larger $p(f(a), g(h(b)), f(c))$ which might be difficult to generate by semantic hyper-linking. $\square$

In the following sections, we will discuss how to extend the basic idea of rough resolution to a powerful companion of semantic hyper-linking to deal with the "large" parts of the proofs.

5.2.1 Maximal Literals

In this section, we show how to identify large literals to resolve upon in rough resolution. Recall that binary resolution on two clauses $C1$ and $C2$ chooses one literal L in $C1$ and one literal M in $C2$ such that $L\theta = \neg M\theta$, where θ is a most general unifier of L and $\neg M$. A resolvent $R{=}(C1 - L)\theta \cup (C2 - M)\theta$ is generated. R is *smooth* if $L\theta\sigma$ is never larger than any literal in $R\sigma$ for any σ; R is *rough* if for some σ, $L\theta\sigma$ is larger than any literal in $R\sigma$. Smooth resolvents are not kept because they only remove small literals (the "smooth" parts of clauses); large literals (the "rough" parts of clauses) remain as a difficulty if R is used in semantic hyper-linking. Thus we are particularly interested in obtaining rough resolvents because they might remove large literals needed for the proof. Rough resolvents can be obtained by resolving upon *maximal literals*.

Definition 70. A literal L in a clause C is an *absolute maximal literal* if, for *all* σ, $L\sigma$ is larger than or equal to that of any literal in $C\sigma$; L is a *likely maximal literal* if for *some* σ, $L\sigma$ is larger than or equal to any literal in $C\sigma$.

A clause can only have one of those two kinds of maximal literals.

Example 27. In clause

$$\{\, d(v, k(w, u)), \neg d(v, u), \neg d(v, w) \,\}$$

where u, v and w are variables, $d(v, k(w, u))$ is the only absolute maximal literal; in clause

$$\{\, d(u, q(w, v)), \neg d(f(u, v), w), \neg d(v, w) \,\}$$

both $d(u, q(w, v))$ and $\neg d(f(u, v), w)$ are absolute maximal literals; clause

$$\{\, d(u, v), \neg d(v, u) \,\}$$

has two absolute maximal literals; in clause

$$\{\, d(u, w), \neg d(u, v), \neg d(v, w) \,\}$$

all three literals are likely maximal literals. □

Definition 71. A *rough resolution step* involves simultaneously resolving upon maximal literals $L_1, \ldots, L_n$ of a clause C (called *nuclei*) with *absolute* maximal literals in other clauses $C_1, \ldots, C_n$ (called *electrons*). A *rough resolvent* is obtained from a rough resolution step.

Nuclei can have absolute or likely maximal literals; electrons can only have absolute maximal literals. Absolute maximal literals in a nucleus are all resolved upon at the same time in one single rough resolution step. This is based on the observation that absolute maximal literals should be eventually all removed since they are always the largest in any instances, and intermediate resolvents might not be kept due to non-negative growths (to be discussed in next section).

Resolving on likely maximal literals in a clause is difficult to handle because it might not delete large literals and, at the same time, could generate too many useless resolvents. For example, the transitivity axiom

$$\neg(x \leq y) \vee \neg(y \leq z) \vee (x \leq z)$$

contains three likely maximal literals and often generates too many clauses during a resolution proof. The role of likely maximal literals is obscure in rough resolution. We use two strategies to set restrictions on applying rough resolution on likely maximal literals.

The first strategy is to apply the following heuristics to simultaneously resolve on more than one likely maximal literal: If there are two likely maximal literals in a clause, we resolve upon them one at a time. This is based on the observation that in most ground instances of such clauses, there are only one absolutely large literal. If there are more than two likely maximal literals, we also resolve upon each possible pair of two of them at the same time. This is based on the observation that usually only few likely maximal literals will become the largest in the ground instances.

Another important strategy is to require that any likely maximal literal resolved upon should still be a likely (or absolute) maximal literal after the proper substitution is applied. Otherwise the resolvent is discarded because it introduces larger literals from those not resolved upon.

Example 28. Let us consider the clause

$$C = \{ \neg p(x, y), \neg p(y, z), p(x, z) \}$$

where all literals are likely maximal literals. Suppose first two literals are resolved with $p(x, a)$ and $p(a, z)$ respectively with substitution $\theta = \{ y \to a \}$. Such resolution is not allowed since none of $p(x, a)$ and $p(a, z)$ are maximal literals in $C\theta$, and the literal not resolved upon, $p(x, z)$, becomes an absolute maximal literal in $C\theta$. □

By carefully choosing maximal literals to resolve upon, we apply resolution in a restricted way (compared with ordinary resolution). However, there are still quite a few resolvents generated. Thus we also need to further choose resolvents to retain, or to delay using some resolvents in further rough resolution. In the next section, we will discuss how to retain resolvents based on the literal sizes.

5.2.2 Retaining Resolvents

Rough resolution only resolves upon maximal literals. Since usually there are not many absolute maximal literals in a clause, the number of resolvents is greatly reduced. However there are still too many resolvents if no further restriction is applied. In this section we discuss one strategy that we use to retain resolvents more selectively.

From the rough resolvents, we prefer those *smaller* than the parents clauses. Ordering on clauses is needed here. Note the literal ordering we have chosen ($\prec_s$) is based on the the numbers of symbols in literals. For clauses, ordering based on the maximum literal size is too weak to be useful since there is not enough distinction among clauses with the same maximum literal sizes. Instead, we extend the ordering based on the multisets of all literal sizes in a clause.

Definition 72. A *multiset* is an unordered collection of possible duplicate elements.

Definition 73. $|C|$ is the multiset of sizes of all literals in C. Difference $|C_1| - |C_2|$ is $c1 - c2$ where $c1$ and $c2$ are the largest elements (0 if the multiset is empty) in $|C_1|$ and $|C_2|$ respectively after common elements are deleted.

For example, for clause $C = \{ p(x), p(y), \neg q(x, y) \}$, $|C| = \{ 3, 2, 2 \}$; $\{ 3, 2, 2 \} - \{ 3, 2 \}$ $= \{ 2 \} - \{ \} = 2$ and $\{ 4 \}$-$\{ 5, 2 \} = -1$.

Definition 74. Suppose in a rough resolution step, resolvent R is obtained from clauses $C_1, \ldots, C_n$. The *growth* of R is $|R|-$ maximum of $(|C_1|, |C_2|, \ldots, |C_n|)$.

Such multiset idea is only used to compute *growth* of a rough resolvent, and the multiset concept can be extended for any ordering on literals. However, in other discussions of this thesis (such as instances generation and UR resolution), we still assume the ordering of clauses in based on the largest literals.

Growth is a useful measurement to retain resolvents from resolving upon likely maximal literals. Intuitively growth indicates the size growth of a rough resolvent relative to the parent clauses *before* substitution is applied. Intuitively, if the growth is negative, the resolvent is "smaller" than the largest parents clause and progress has been made to reduce the number of large literals. On the other hand, if the growth is zero or positive, the resolvent is of the same length as or larger than the largest parent clause. There is no progress from this rough resolution step and, from the viewpoint of semantic hyper-linking, it is not useful to keep the resolvent. Thus they are retained only when there are no other resolvents with negative growths. The effect is that we delay the use of such large (compared with the parent clauses) resolvents in further rough resolution. Intuitively this can help reduce the number of clauses used in rough resolution so smaller ones can be used sooner in semantic hyper-linking. It also avoid large ones being used early in rough resolution to derive larger ones of which ground instances will never be generated.

The algorithm of rough resolution is described in Fig. 5.1. Resolvents of non-negative growth are retained only when there are no resolvents with negative growth. The procedure repeats until a proof is found. The resolvents are used in semantic hyper-linking in a limited way because many resolvents could be generated. Reasonable time bound is set on the use of rough resolvents in semantic hyper-linking.

The collaboration of rough resolution and semantic hyper-linking is not explicitly shown in Fig. 5.1. Basically rough resolution executes for some amount of time then stops, and new resolvents are used in later semantic hyper-linking; when executed again, rough resolution picks up from where it left off and continues. In Section 8.2 we will discuss such collaboration in more detail.

The rough resolvents with negative growth are always kept; among the rough resolvents with non-negative growth, only the smallest (in terms of the largest literal

```
Algorithm Rough Resolution
begin
  loop
    for each clause C with absolute maximal literals
      Obtain a new rough resolvent R using C as nucleus
      if R has a negative growth
      then
        save R permanently as a new clause
      else
        save R temporarily
  until there are no new resolvents with growth < 0
  for each clause C having likely maximal literals
    loop
      Obtain a new rough resolvent R using C as nucleus
      if R has negative growth
      then
        save R permanently as a new clause
      else
        save R temporarily
    until no new rough resolvents can be generated
  if in last loop no rough resolvent was generated
        with negative growth
  then
    for each smallest temporarily saved rough
          resolvent R
      save R permanently as a new clause
end
```

Figure 5.1: Rough resolution algorithm

in the resolvent) are saved, if necessary. This allows the resolvents with non-negative growth to be used in a controlled manner.

With restrictions on how rough resolution is applied (by resolving upon maximal literals simultaneously) and how resolvents are retained (based on growth), much less resolvents are retained than those in other similar resolution strategy. And we have found the algorithm practical and useful when used with semantic hyper-linking.

5.3 An Example

Bledsoe gave LIM+ problems in [Ble90] as challenge problems for automated theorem provers. Because of the large search space they might generate, LIM+ problems are difficult for most theorem provers. However, they are not difficult for Str$\dot{+}$ve [Hin92] which has built in inequality inference rules for dense linear ordering.

In this section we will look at the proof of LIM3 using rough resolution. Intermediate results from semantic hyper-linking are omitted.

As mentioned in the introduction, the correct ground instance of the goal clause has to be generated to obtain the proof. However, that ground instance contains a literal so large that semantic hyper-linking cannot generate it early enough in the search for the proof. As a result, the prover gets lost even before correct goal instances are generated. Rough resolution helps to delete large literals by resolving upon them; smaller ground literals are generated by semantic hyper-linking. It is interesting to observe that the proof presented here does not need the term $min(d1(ha(e0)), d2(ha(e0)))$ which is essential to human proofs.

We only list clauses used in the proof; literals in boxes are maximal literals and only clauses 14 and 15 (from the same transitivity axiom) have likely maximal literals. Each rough resolution step is denoted by a list of clause numbers, with nuclei in boxes. For example, ($\boxed{14}$,3,17) denotes a rough resolution step using clause 14 as nucleus and clauses 3 and 17 as electrons.

Input clauses:

3: $\{\ \boxed{lt(ab(pl(X,Y)), pl(ab(X), ab(Y)))}\ \}$

8: $\{\ \boxed{lt(ab(pl(f(Z), ng(f(a)))), E)}, lt(E, o),\ \neg lt(ab(pl(Z, ng(a))), d1(E))\ \}$

9: $\{\ \boxed{lt(ab(pl(g(Z), ng(g(a)))), E)}, lt(E, o),\ \neg lt(ab(pl(Z, ng(a))), d2(E))\ \}$

10: $\{\ \boxed{lt(ab(pl(xs(D), ng(a))), D)}, lt(D, o)\ \}$

13: $\{\ \boxed{lt(X,Y)}, \boxed{lt(Y,X)}\ \}$

14: $\{\ \boxed{\neg lt(X,Y)}, \boxed{\neg lt(Y,Z)}, lt(X,Z)\ \}$

(14 and 15 are the same except for maximal literals chosen for resolution)

15: $\{\ \boxed{\neg lt(X,Y)}, \boxed{lt(X,Z)}, \neg lt(Y,Z)\ \}$

17: $\{\ \boxed{lt(pl(X,Y),Z)}, \neg lt(X, ha(Z)), \neg lt(Y, ha(Z))\ \}$

18: $\{\ lt(D,o),\ \boxed{\neg lt(ab(pl(pl(f(xs(D)), ng(f(a))), pl(g(xs(D)), ng(g(a))))), e0)}\ \}$

The proof:

21: ($\boxed{14}$,3,17) $\{\ \boxed{lt(ab(pl(X,Y)),Z)}, \neg lt(ab(X),ha(Z)),$
$\neg lt(ab(Y),ha(Z))\ \}$

24: ($\boxed{18}$,21) $\{\boxed{\neg lt(ab(pl(f(xs(D)),ng(f(a)))),ha(e0))}$
$\boxed{\neg lt(ab(pl(g(xs(D)),ng(g(a)))),ha(e0))}, lt(D,o)\}$

29: ($\boxed{24}$,8,9) $\{\boxed{\neg lt(ab(pl(xs(D),ng(a))),d1(ha(e0)))},$
$\boxed{\neg lt(ab(pl(xs(D),ng(a))),d2(ha(e0)))},$
$lt(ha(e0),o), lt(D,o)\}$

97: ($\boxed{15}$,29,10) $\{\boxed{\neg lt(ab(pl(xs(D),ng(a))),d2(ha(e0)))},$
$\neg lt(D,d1(ha(e0))), lt(ha(e0),o), lt(D,o)\}$

99: ($\boxed{15}$,29,10) $\{\boxed{\neg lt(ab(pl(xs(D),ng(a))),d1(ha(e0)))},$
$\neg lt(D,d2(ha(e0))), lt(ha(e0),o), lt(D,o)\}$

139: ($\boxed{10}$,97) $\{\ \boxed{\neg lt(d2(ha(e0)),d1(ha(e0)))},$
$lt(ha(e0),o), lt(d2(ha(e0)),o)\ \}$

($lt(ha(e0),o)$ and $lt(d2(ha(e0)),o)$ are then unit deleted)

141: ($\boxed{10}$,99) $\{\ \boxed{\neg lt(d1(ha(e0)),d2(ha(e0)))},$
$lt(ha(e0),o), lt(d1(ha(e0)),o)\ \}$

($lt(ha(e0),o)$ and $lt(d1(ha(e0)),o)$ are then unit deleted)

($\boxed{13}$,139,141) $\{\ \}$ is obtained and a proof is found.

Unit $\{\neg lt(ha(e0),o)\}$ is generated by model filtering (see Chapter 4) in semantic hyper-linking. Or it can be generated by UR resolution from clause 4 and 7. It is then used with clause 5 to generate $\{\neg lt(d1(ha(e0)),o)\}$; with clause 6 to generate $\{\neg lt(d2(ha(e0)),o)\}$.

4: $\{\neg lt(e0,o)\}$

5: $\{lt(X,o), \neg lt(d1(X),o)\}$

6: $\{lt(X,o), \neg lt(d2(X),o)\}$

7: $\{lt(X,o), \neg lt(ha(X),o)\}$

None of the above three units can be generated by rough resolution. This shows the collaboration of rough resolution with semantic hyper-linking to find the proof.

Problem	*with rough resolution*	*without rough resolution*
Am8	2127.3	—*
LIM1	83.0	—
LIM2	63.3	—
LIM3	534.8	—
IMV	374.5	49.8
Ipc I1	1.9	344.6
Ipc IP1	29.2	1614.1
Ipc P1	65.8	—
Ipc S1	2.9	805.4
ls37	138.7	65.2
SAM's Lemma	146.5	95.5
wos15	282.7	—
wos19	407.5	1183.2
wos21	783.7	93.4
wos31 (EXQ1)	203.5	39.2
EXQ2	84.0	130.1
EXQ3	140.3	112.0

*"—" indicates the run is aborted after either running over 20,000 seconds or using over 30 Megabyte memory

Table 5.1: Proof results using rough resolution with semantic hyper-linking

5.4 Experimental Results

In this section we give experimental results, with or without rough resolution, on some hard problems. Those experimental results show that rough resolution indeed improves the prover and often helps to find proofs faster. We also feel it is possible that a slightly different rough resolution (for example, one with a different criterion in resolving on likely maximal literals) could help a lot. Table 5.1 lists some results (in seconds) that show rough resolution is in general a useful technique used with semantic hyper-linking. The descriptions of the problems can be found in the Appendix A.

LIM1–3 are three simpler Lim+ problems proposed in [Ble90]. Prover STR$\dot{+}$VE, with built in inference rules for dense linear ordering, can easily prove all Lim+ problems. However, few other general-purpose theorem provers can prove LIM1–3 (especially LIM3). METEOR [AL91] can prove all three by using special guidance.

OTTER [McC90] and CLIN [LP92] could not prove any Lim+ problem.

5.5 Completeness and Soundness

Rough resolution is sound since it is basically resolution with restrictions. The resolvents by rough resolution are logical consequences of the input clauses. Thus semantic hyper-linking augmented with rough resolution is still sound, namely, if a proof is found for an input clause set S, S is unsatisfiable.

Rough resolution, however, is incomplete due to various restrictions on application of resolution and retaining resolvents. Therefore rough resolution should be used with some complete method to focus on solving the "large" part of the proof. In this regard, we are interested in the completeness of using rough resolution with semantic hyper-linking—the combined method should remain complete.

We now prove a more general completeness theorem about combining semantic hyper-linking and resolution (or any refinement) which executes for a finite number of steps in each round of semantic hyper-linking. Resolvents are treated as input clauses and ground instances are generated from resolvents and input clauses. Combining resolution with semantic hyper-linking this way will still find the proof if there is one.

Theorem 75. *If in each round of semantic hyper-linking, resolution is executed for a finite number of steps and minimal ground instances are generated from resolvents and input clauses, then the combined method is still complete.*

Proof. Suppose the input clause set is S and unsatisfiable. Let the set of resolvents generated by resolution is R. Then $S \cup R$ is also unsatisfiable. Since semantic hyper-linking is complete, semantic hyper-linking on $S \cup R$ will terminate with a proof. □

In general, rough resolution is useful when used with other complete methods that prefer "small" clauses. It helps to focus on removing "large" part of the proofs. In particular, we have found that semantic hyper-linking and rough resolution conceptually work well together: semantic hyper-linking solves the "small" and "non-Horn" part of the proof; UR resolution [CP93b] solves the "Horn" part of the proof; and rough resolution solves the "large" part of the proof.

Chapter 6

The Use of Semantics

Semantic hyper-linking uses semantics (the input structure I) to control the proof search in two ways: First, I is explicitly expressed by *evaluation procedures* (for ground terms and literals) and is used to construct new structures $I[\mathcal{L}]$ during the search for the proofs so the new structures need not be explicitly expressed. Second, if I is decidable, I can be used to prune the search space in instances generation by discarding partial hyper-links that have no instances false in $I[\mathcal{L}]$ so such fruitless search is avoided (see Section 3.3). In this case, the semantics has to be expressed by *decision procedures* as well for the underlying domain to detect the validity of non-ground terms and literals. Also, decidable semantics can be used naturally in determining the *support* status of a literal or a clause, which provides a mechanism similar to the set of support strategy [WRC65].

Example 29. Suppose we have input clauses containing predicate $p(x, y)$ for $x < y$, function $f(x, y)$ for $x + y$, constants a for 1 and b for 2. Suppose the domain is the set of real numbers. Then we can easily define evaluation procedures to evaluate ground literals such as $p(f(a, b), a)$ to be **F** (since $1 + 2 < 1$), and $p(a, f(a, b))$ to be **T** (since $1 < 1 + 2$). It is not possible to evaluate non-ground $p(f(x, a), y)$ because of the variables x and y. However, one can decide if $p(f(x, a), y)$, equivalently $x + 2 < y$, is valid using the decision procedure Sup-Inf due to Bledsoe [Ble75, Sho77] (to be discussed in Section 6.3.2) or a more general decision procedure due to Presburger or Cooper [Pre29, Coo72, Coo71]. $\square$

In fact, the input semantics I does not have to have a decision procedure. It suffices that any result reporting validity is indeed sound. Semantic hyper-linking is still complete as long as there is a procedure P for I such that for any possibly non-ground clause C, P always terminates on C and if P determines that C is valid in I, it is true that $I \models C$. It does not hurt if P fails to determine validity on all possible C. This is because P is mainly used to discard valid partial hyper-links to gain efficiency in ground instance generation. P is not needed for completeness. Note, however, in this case, P cannot be used to compute the support status of a non-ground literal or clause.

In general, once the semantics is given, it is possible to determine the evaluation procedures since they are just ordinary program procedures. The decision procedures, however, do not always exist depending on the underlying semantics. In the next section, we will discuss how to design semantics for a given first-order formula. In the sections following that, we will discuss the use of various kinds of semantics with semantic hyper-linking. We will also show how multiple semantics can be used, and discuss how different semantics effect the efficiency in search of proofs.

6.1 Designing Semantics

In general, the input semantics (structure) is actually a specific *example* or *model* for the related axioms and theorem. Also, because of the Skolem functions and constants, the semantic information should be designed for the clauses, not just the original first-order formulas. In this regard, designing semantics is not a simple task since clauses are often very different from the original formulas and often the Skolem functions and constants (introduced from the Skolemization process) have obscure meanings. Moreover, in cases where standard Skolemization introduces too many clauses, one often has to use (non-standard) structure-preserving[1] clauses translation [PG86] which might also introduce new predicate symbols.

[1]Here the structure indicates the syntactic form of the original first-order formula.

6.1.1 Skolem Function and Constants

Skolemization introduces Skolem functions and constants when converting a first-order formula into a set of clauses. If a Skolem function comes from the original formula naturally, we usually can find a proper meaning for it. For example, a Skolem function, say $s(x)$ might be introduced to replace y when we transform the formula $\forall x \exists y (x \leq y)$ into a clause. We can define $s(x) = x + 1$. In such cases, Skolem functions often are like other functions in the formulas and have natural semantics.

However, there are cases that it is hard to interpret the Skolem functions, especially when Skolem functions are introduced by negations of theorems which convert the universal and existential quantifiers.

Example 30. The following is the AM8 theorem (see Appendix A):

$$\exists x (a \leq x \leq b \wedge \forall y (a \leq y \leq b \supset f(x) \leq f(y)))$$

With $p(x, y)$ to denote $x \leq y$, it becomes:

$$\exists x (p(a, x) \wedge p(x, b) \wedge \forall y ((p(a, y) \wedge p(y, b)) \supset p(f(x), f(y))))$$

We have to negate this theorem before we submit it to a refutation theorem prover. The negation of the theorem in Skolemized disjunctive normal form is:

$$\neg p(a, x) \vee \neg p(x, b) \vee (p(a, k(x)) \wedge p(k(x), b) \wedge \neg p(f(x), f(k(x))))$$

in which a Skolem function $k(x)$ is introduced with an obscure meaning. $\square$

Wang [Wan85] used an *interpretation normal form (IPN)* to help determine the semantics of the Skolem functions (or any function with unknown semantics) if their semantics are not available. In addition, IPN can help check the correct semantics of other functions and predicates. IPN is mainly used to design a semantics (example) such that semantically guided hierarchical deduction, which is not complete, does not fail. A similar idea has been useful in designing semantics of many problems for semantic hyper-linking.

An interpretation normal form is basically a set of selected clauses, $C_1 \wedge \ldots \wedge C_h$, from the clause set obtained by Skolemization. The basic idea of using IPN is this: After a natural semantics is constructed (if none can be found, we can always use trivial semantics. See Section 6.4) for the symbols occurring in the original first-order formula, we are left with some Skolem functions or constants that have unknown semantics. After identifying such Skolem functions or constants in a clause set S, and, for each such function or constant symbol, combining all clauses of S containing an occurrence of this symbol into an IPN form, then one can interpret the symbol.

The interpretation rule is simple: Given an IPN $C_1 \wedge \ldots \wedge C_h$, we first interpret those with known semantics. Then for each C_i in the IPN if C_i is a part of the goal clause, we make C_i false; otherwise we make C_i true.

Such task seems simple but is difficult to carry out, especially when the clause in a IPN has more than one unknown Skolem functions or constants.

6.1.2 Non-Standard Clause Form Translation

Structure-preserving clause translation [PG86] is a non-standard clause form translation (compared with standard Skolemization procedure) that preserves more of the (syntactic) structure of the formula than the Skolemization procedure. Also, such non-standard clause form translation avoids the exponential increase in size which may occur with the Skolemization procedure. In particular, it is useful when Skolemization generates too many clauses for theorem provers to handle (see the example below). For resolution style theorem provers, sometimes formulas such as $A \wedge \neg A$ where A is a large sub-formulas, are difficult to prove because A or $\neg A$ is split into many clauses which only contain part of both formulas. Non-standard clause form translation was introduced to cope with this problem and enable the resolution style theorem provers to take advantage of such structural properties.

The basic idea of the method is to introduce new propositional predicate P_A to refer to various sub-formulas A of the original formula. Then A is replaced by P_A everywhere and $P_A \equiv A$ is added. Fortunately in many cases, only $P_A \supset A$ or $A \supset P_A$ is needed. Thus sub-formulas (in the form of new predicates) occur naturally in the resulting clauses and the proofs.

Example 31. The Andrews' challenge problem [Pel86] is the following short first-order formula:

$$
\begin{aligned}
& ((\exists x \forall y(p(x) \equiv p(y))) \equiv (\exists x(q(x)) \equiv \forall y(q(y)))) \\
\equiv \quad & ((\exists x \forall y(q(x) \equiv q(y))) \equiv (\exists x(p(x)) \equiv \forall y(p(y))))
\end{aligned}
$$

However, the ordinary Skolemization process translating the formula to clauses, generates over 1,600 clauses and introduces many Skolem functions[2]. With the non-standard clause translation process [PG86], this problem generates far less (35) clauses.

```
[not l11, q(V1)].
[l11, not q(sc1)].
[not l10, p(sc1)].
[l10, not p(V1)].
[not l9, p(V1)].
[l9, not p(sc1)].
[not l8, q(sc1)].
[l8, not q(V1)].
[not l7, not l10, l11].
[not l7, l10, not l11].
[l7, not l10, not l11].
[l7, l10, l11].
[not l6, not q(V1), q(sc1)].
[not l6, q(V1), not q(sc1)].
[l6, not q(sf1(V2)), not q(V2)].
[l6, q(sf1(V2)), q(V2)].
[not l5, not l8, l9].
[not l5, l8, not l9].
[l5, not l8, not l9].
[l5, l8, l9].
[not l4, not p(sc1), p(V1)].
[not l4, p(sc1), not p(V1)].
```

[2] With $A \equiv B$ carefully replaced by $(A \wedge B) \vee (\neg A \wedge \neg B)$ instead of $(A \supset B) \wedge (B \supset A)$, the number of clauses is reduced to 127.

```
[l4, not p(V2), not p(sf1(V2))].
[l4, p(V2), p(sf1(V2))].
[not l3, not l6, l7].
[not l3, l6, not l7].
[l3, not l6, not l7].
[l3, l6, l7].
[not l2, not l4, l5].
[not l2, l4, not l5].
[l2, not l4, not l5].
[l2, l4, l5].
[not l2, not l3, l1].
[l2, l3, l1].
[not l1].
```

Although only one Skolem function (`sf1`) and one Skolem constant (`sc1`) are introduced, the clause set introduces eleven new predicates (`l1`–`l11`). □

Non-standard clause translation is useful for semantic hyper-linking in two aspects: First, sometimes it reduces the number of clauses generated. Second, it often only introduces propositional predicates which make the resulting clause sets close to "propositional." Semantic hyper-linking is good at such problems. However, determining the natural semantics is not a trivial task (discussed in Section 6.4).

6.2 Finite Semantics

A finite semantics (structure) has a finite domain. The decision procedures on a non-ground literal L can always be implemented using the evaluation procedures on all possible ground instances of L by systematically substituting the variables in L with the constants in the domain. Thus decision procedures always exist for finite semantics and often can be realized by the evaluation procedures. Thus each finite semantics with evaluation procedures, is decidable.

Example 32. Suppose we have an input clause set in group theory. Suppose the semantics is a finite group $\mathbf{Z}_3$ (with domain values 0, 1, 2) the operation is addition modulo 3. And suppose we have a predicate $p(x, y, z)$ for $x * y = z$, $g(x)$ for x^{-1}. Then to test $p(g(x), g(y), x)$ for satisfiability, one can substitute x and y with the three domain values (nine cases in total) and explicitly ground the literal and evaluate using evaluation procedures. It is easy to see the literal is true with the substitution $\{x \leftarrow 0, y \leftarrow 0\}$. Also, for the ground literal $p(g(2), g(1), 2)$, it can be evaluated as $2^{-1} * 1^{-1} = 2$ which is true since in $\mathbf{Z}_3$, it is true that $2^{-1} = 1$, $1^{-1} = 2$ and $1 * 2 = 2$. □

Note, however, testing all possible ground instances of literal L could be computationally expensive. Suppose the number of variables in L is v and the domain has a size n, the cases to test is at most n^v. And the experiments show that with larger domains, the semantic tests usually are expensive. In some cases, such tests are too slow to be practically useful. Thus semantics with smaller domain is always desirable. However, big domains may help sometimes since they introduce less equalities that do not hold in general.

6.3 Infinite Semantics

An infinite semantics has an infinite domain so enumerating ground instances of a non-ground literals is not possible. Thus a decision procedure is not always available except for *decidable semantics*. For decidable semantics, such decision procedures exist and efficient pruning of the search space is possible. For *undecidable semantics*, such as the whole of real number arithmetic (including exponentiation and trigonometry) which is undecidable, the instance generation stage might need to search through a lot of partial hyper-links before proper ground instances are generated. This often suffers from inefficiency. Note, however, semantic hyper-linking is still complete in this case if there are evaluation procedures—decision procedures only have an effect on efficiency.

6.3.1 Decidable Semantics

Recall that for a decidable structure I, given a clause C which possibly contains variables, it is decidable if $I \models C$. For infinite decidable structure, this requires decision procedures for validity. In general, any decidable semantics can be used with semantic hyper-linking. In this section, we will discuss the use of decidable semantics in a common context in arithmetic.

6.3.2 Bledsoe Real Arithmetic and Sup-Inf Procedure

Complete real arithmetic, involving operations such as exponentiation and trigonometry, is not decidable. However, a reasonable subset of complete real arithmetic is decidable. And often such subset is expressively powerful enough.

Definition 76 ([Ble75]). An expression is said to be a formula in *Presburger arithmetic* if it is a well formed algebraic expression, allowing only variables, integer constants, addition and subtraction, constant multiplication, the arithmetic relations $\leq$ and $=$, the logical connectives (such as $\vee, \wedge$, and $\neg$), and quantification (either universal or existential).

Example 33. The following formulas are some examples of expressions in Presburger arithmetic:

$$\forall x \exists y (3x + y = 2 \supset x < y$$
$$\forall x \forall y [(x < 3y + 2) \wedge x = 1 \supset x = y]$$
$$\forall x \exists y \forall z [(y + z < 5 \wedge 3z \geq y + 8 \wedge x \leq y) \supset (2 + z > 2y \vee x > z - 5)]$$

□

Presburger [Pre29] first shows decision procedures exist for Presburger arithmetic (hence the name). Other known decision procedures can be found in [Coo71, Coo72]. The basic idea of such decision procedures is based on a method of elimination of quantifiers. However, in general such algorithms tend to generate a combinatorial expansion, and are not practical. In [Opp75] it is shown that in the worst case, the time complexity of decision procedures for Presburger arithmetic is at least $O(2^{2^{2^n}})$

in the length of the formula. Although such worse cases seldom happen in practice, it still seems impractical to choose Presburger arithmetic as the semantics because of the expensive semantic checks.

In [Ble75] Bledsoe propose a decision procedure, the *Sup-Inf* procedure, for a subclass of Presburger arithmetic that allows only universally quantified variables which take integer values. Although relatively smaller compared with full Presburger arithmetic, such a subclass is practically useful, especially in program verification and related applications. Additionally, the Sup-Inf procedure is bounded above by 2^n, so it is much easier and faster than full Presburger arithmetic. Bledsoe also proves that Sup-Inf determines validity—a formula F determined to be valid is indeed valid; otherwise F may or may not be valid. Although this makes Sup-Inf a semi-decision procedure that only decides validity (but terminates in any case), it suffices to work with semantic hyper-linking: Suppose that we generate a partial hyper-link $C\alpha = C_1\alpha \cup C_2\alpha$ with $C_1\alpha$ hyper-linked with eligible literals and $C_2\alpha$ ground instances to be generated. If $C_2\alpha$ is determined valid by Sup-Inf, $C\alpha$ can be safely discarded since Sup-Inf decides validity, otherwise $C_2\alpha$ might be valid or invalid, and in this case $C\alpha$ is kept and used in ground instance generation.

In [Sho77] Shostak proves that the Sup-Inf procedure by Bledsoe calculate exact bounds for, in real values, a variable. Shostak also improves and extends the Sup-Inf procedure so it determines invalidity as well as validity (thus complete) over a somewhat larger domain than Bledsoe's method. This improved method, however, is still a semi-decision procedure for the subclass of Presburger arithmetic identified by Bledsoe.

Since Sup-Inf calculates upper and lower bounds in real numbers, it can be extended for variables taking real values. Thus real constants can also be used. In [Bun83] Bundy calls such extended subclass *Bledsoe real arithmetic*.

Definition 77 ([Ble75, Bun83]). *Bledsoe real arithmetic* is an extended subclass of Presburger arithmetic that permits real constants (not just integer), but allows only universally quantified variables (existentially quantified variables cannot be present).

Example 34. The following examples are in Bledsoe real arithmetic and Presburger arithmetic:

$$\forall x \forall y[(x < 3y + 2) \wedge x = 1 \supset x = y]$$
$$\forall x \forall y \forall z[(y + z < 5 \wedge 3z \geq y + 8 \wedge x \leq y) \supset (2 + z > 2y \vee x > z - 5)]$$

However, the following are only in Bledsoe real arithmetic because of the real constants:

$$\forall x \forall y(x + 0.5y < 5)$$
$$\forall x \forall y \forall z[(x - 1.2y < z) \supset (0.5z + 0.3y < x)]$$

Note the essential differences between full Presburger and Bledsoe real arithmetic are the universal quantification and real constant multiplication. □

The basic idea of Sup-Inf is calculating the upper (using the Sup procedure) and lower (using the Inf procedure) bounds (in the domain of real numbers) of each variables (probably in terms of other variables). A recursive procedure is used to detect the solvability of those bounds. We now outline the process determining the validity of a formula F using Sup-Inf.

Given a formula F in Bledsoe real arithmetic, the Sup-Inf procedure is carried out in two stages. In the first stage the formula F is converted to a set of (disjuncted) linear programming problems with the property that F is valid if and only if none of the problems has a solution. In the second stage, each linear programming problem is tested for solvability using the Sup and Inf procedures.

Here is a close look at Sup-Inf. In the first stage, the formula F is negated and converted into a disjunction normal form:

$$\neg F \equiv G_1 \vee G_2 \vee \ldots \vee G_p$$

where each G_i is a conjunction of linear inequalities of the form $A \leq B$. During the conversion, terms of the form $A = B$ are removed and every A is replaced by B in the corresponding G_i. Also, $A \geq B$ is replaced by $B \leq A$, $\neg(A \leq B)$ by $B < A$, $\neg(A = B)$ by $(A < B \vee B < A)$, and $A < B$ by $A + \epsilon \leq B$ where ϵ is a new *infinitesimal* constant that is non-negative yet smaller than any positive real number. The use of ϵ is the major difference in [Bun83] from earlier methods [Ble75, Sho77].

Essentially the use of ϵ allows calculation of bounds for real variables. Since $\neg F$ has all the variables existentially quantified, we need to check the solvability of each G_i.

Example 35. Suppose the following formula is given:

$$F \equiv (x < 4y + 2.5) \wedge (x = y + 2) \wedge (4.2x \leq 3y + 2.2) \supset x = y.$$

Then

$$\begin{aligned}
\neg F &\equiv \neg[x < 4y + 2.5 \wedge (x = y + 2) \wedge (4.2x \leq 3y + 2.2) \wedge x = y] \\
&\equiv x < 4y + 2.5 \wedge (x = y + 2) \wedge (4.2x \leq 3y + 2.2) \wedge (x < y \vee y < x) \\
&\equiv (x < 4y + 2.5 \wedge (x = y + 2) \wedge (4.2x \leq 3y + 2.2) \wedge x < y) \\
&\quad \vee(x + \epsilon \leq 4y + 2.5 \wedge (x = y + 2) \wedge (4.2x \leq 3y + 2.2) \wedge y < x) \\
&\equiv (y + 2 + \epsilon \leq 4y + 2.5 \wedge (4.2(y + 2) \leq 3y + 2.2) \wedge x + \epsilon \leq y) \\
&\quad \vee(y + 2 + \epsilon \leq 4y + 2.5 \wedge (4.2(y + 2) \leq 3y + 2.2) \wedge y + \epsilon \leq x).
\end{aligned}$$

so we can get

$$G_1 = \{\, y + 2 + \epsilon \leq 4y + 2.5, 4.2(y + 2) \leq 3y + 2.2, x + \epsilon \leq y \,\}$$

and

$$G_2 = \{\, y + 2 + \epsilon \leq 4y + 2.5, 4.2(y + 2) \leq 3y + 2.2, y + \epsilon \leq x \,\}.$$

$\neg F$ is satisfiable if and only if either G_1 or G_2 has a solution in reals, and so F is valid if and only if neither G_1 nor G_2 has solutions. □

In the second stage, two recursive procedures, Sup and Inf, are used to calculate the bounds for each variable v: Sup computes the maximum value v can take in real solutions (the upper bound), and Inf computes the minimum value v can take in real solution (the lower bound). The solvability can easily be determined by recursively checking the bounds of variables. Detailed descriptions of the Sup and Inf procedures can be found in [Ble75, Sho77, Bun83].

6.4 Trivial Semantics

Semantic hyper-linking needs an initial (input) semantics to search for the proof. Semantic information is needed for all the constants, functions and predicates in the input clauses. However, natural semantics is not always available. This is not unusual since structural and semantic information of the original first-order formulas, often is lost after Skolemization (converting a first-order formula into a set of clauses), partly due to excessive Skolem functions and Skolem constants introduced during the process. Therefore, if one has no access to the original first-order formula (especially in the theorem proving community people tend to circulate problems in clause form only), it is not trivial to design proper natural semantics for the clauses.

Example 36. For problems like Andrews' challenge problem (see Example 31), too many clauses are generated with new Skolem functions. It is difficult, if not impossible, to determine the semantics of those Skolem functions. Although non-standard clause translation generates far less clauses with much fewer Skolem functions and constants, due to the new predicates introduced during the translation process, a natural semantics for the clauses is still difficult to get even the original formula is given. In practice it is easier to use trivial semantics for the theorem prover to prove the corresponding clause set. Recall that a trivial semantics is one in which all atoms in the Herbrand base are true (or false). □

Recall that the completeness of semantic hyper-linking is independent of the initial (input) semantics. In other words, one can choose any semantics as input. This relieves one of finding a natural semantics when one is not available. Thus one always has the choice of trivial semantics, namely, *not* specifying any semantics at all. Such freedom allows semantic hyper-linking to be evaluated at the same level as other widely used syntactic theorem proving techniques.

In trivial semantics, both evaluation and decision procedures are straightforward and the semantic check is fast. However, when trivial semantics is used, the proof search is not goal-sensitive and the search space is not pruned as by natural semantics. We will discuss this in later section. In Chapter 8 we will see natural semantics, if available, in general helps to find fast proofs.

Natural Semantics	*Approximate Semantics*
$x \mid y$	$x \leq y \wedge x \geq 0$
$gcd(x, y)$	$min(x, y)$
$x * y$	$x + y$
integer quotient of x/y	$x - y, if\ y \leq x; 0\ otherwise$

Figure 6.1: An example of using approximate semantics

6.5 Approximate Semantics

Since semantic hyper-linking is complete and independent of the input semantics, we can use semantics that is not related to the problem to be proved (although the choices of semantics often have influences on the performances). Therefore we can *simulate* the functions not expressible in decidable semantics, by those in decidable semantics such as Bledsoe real arithmetic. The idea is to use another semantic which closely resembles the one in question, and can be expressed using, for example, functions in Bledsoe real arithmetic. The choices are not arbitrary because using irrelevant semantics often makes the performance undesirable. We call the simulating semantics the *approximate semantics*; the semantics being simulated is the natural semantics. To expect better performance using the approximate semantics, approximate and natural semantics should at least behave the same on the input clauses, namely, they should obtain the same set of axioms and goals from the input clauses.

For example, in Fig. 6.1 we show the approximate semantics for *division, integer quotient, multiplication*, and *gcd*. Also, for most set theory functions, we can simulate them using functions on cardinal numbers.

Although approximate semantics is not always available, in many cases we have found such approximation practical enough to allow us to "simulate" many functions in undecidable semantics. Even though approximate semantics seems to be worse than the a natural one, we believe such simulation might be better overall in the search for the proofs.

6.6 The Use of Multiple Semantics

In this section, we discuss, for a given input clause, how to use more than one semantics with semantic hyper-linking.

If, for a given input clause set S, more than one structure (or semantics) can be used with semantic hyper-linking, the input structure is taken as the *intersection* of those structures.

Definition 78. For a clause set S, suppose $I_1, \ldots, I_n$ are Herbrand structures for S with the Herbrand base H. Then the *intersection* $\bigcap I$ of $I_1, \ldots, I_n$ is defined as follows (similar to Definition 41):

(1) An atom $A \in H$ is true in $\bigcap I$ if and only if A is true in every I_i; otherwise, A is false in $\bigcap I$.

(2) A negative literal $\neg A$ (where $A \in H$) is true in $\bigcap I$ if and only if A is false in $\bigcap I$; otherwise, it is false in $\bigcap I$.

(3) A ground clause $L_1 \vee \ldots \vee L_m$ is true in $\bigcap I$ if and only if at least one L_i is true in $\bigcap I$; otherwise, it is false in I.

(4) A clause C is true in $\bigcap I$ if and only if *every* ground instance $C\sigma$ of C is true in $\bigcap I$; otherwise, C is false in $\bigcap I$.

(5) A set S of clauses is true in $\bigcap I$ if and only if each clause in S is true in $\bigcap I$; otherwise, S is false in $\bigcap I$.

Theorem 79. *If each I_i of $I_1, \ldots, I_n$ is consistent, then $\bigcap I$ is also consistent.*

Proof. Assume that $\bigcap I$ is inconsistent. Without loss of generality, we can assume there is an atom A in the Herbrand base such that A and $\neg A$ are both true in $\bigcap I$. Since $\neg A$ is true in $\bigcap I$, according to the definition of $\bigcap I$, there exist one I_k such that $\neg A$ is true in I_k. Also, since A is true in $\bigcap I$, A is true in every I_i. Thus A and $\neg A$ are both true in I_k which contradicts the given facts that all I_i are consistent. A contradiction is derived. Thus the theorem is proved. $\square$

Thus, the intersection $\bigcap I$ of structure $I_1, \ldots, I_n$ is still consistent and can be used as a new structure. Intuitively, as Definition 78 implies, the number of atoms in the Herbrand base that are true $\bigcap I$ is less than or equal to that in any of I_i. Thus $\bigcap I$ might have more "false" ground atoms then any constituent I_i.

Example 37. Suppose for a given clause set S, $I_{\mathbf{F}}$ is the Herbrand structure in which all the atoms in the Herbrand base are false, and $I_{\mathbf{T}}$ is the Herbrand structure in which all the atoms in the Herbrand base are true. Thus the intersection of $I_{\mathbf{F}}$ and $I_{\mathbf{T}}$, $I_{\mathbf{F}} \cap I_{\mathbf{T}}$, is just $I_{\mathbf{F}}$ since every atom in the Herbrand base is now false in $I_{\mathbf{F}} \cap I_{\mathbf{T}}$. □

Now let us consider the use of multiple semantics, represented as the intersection of given structure $I_1, \ldots, I_n$, with semantic hyper-linking. We have to consider the evaluation procedures and decision procedures for $\bigcap I$.

The evaluation procedures for ground cases are straightforward based on conditions (1)–(3) in Definition 78. However, in non-ground cases, the decision procedure is not always possible by combining decision procedures for each I_i. If each I_i is a finite semantics, the decision procedure is again available by following Definition 78 since for a non-ground clause, every instance can be obtained by replacing every variable by proper values in the domain. Thus for finite semantics, their intersection has both evaluation procedures and decision procedures which can be easily obtained from the individual evaluation procedures for each I_i.

On the other hand, if each I_i is an infinite semantics and has a decision procedure, the validity of a non-ground clause C cannot always be detected by using decision procedures for I_i.

Example 38. Given the clause $C = p(x) \vee q(y)$ and two infinite structures I_1 and I_2 for which there exist decision procedures P_1 and P_2 respectively. Suppose C is true in I_1, then for every instance $p(a) \vee q(b)$ of $p(x) \vee q(y)$, at least one of $p(a)$ or $q(b)$ is true in I_1. Suppose also that C is true in I_2, then for every instance $p(a) \vee q(b)$ of $p(x) \vee q(y)$, at least one of $p(a)$ or $q(b)$ is true in I_2. However, it is not guaranteed that $p(x) \vee q(y)$ is true in $I_1 \cap I_2$ which requires that for every instance $p(a) \vee q(b)$, at least one of $p(a)$ or $q(b)$ is true in *both* I_1 and I_2. It could happen that for each instance $p(a) \vee q(b)$, only one of them is true in I_1 and the other is only true in I_2. □

Such a phenomenon is observed by Apt and van Emden [AvE82]. Fortunately, Horn clauses enjoy a property which allows the intersection of models to also be a model. This is the *model intersection property.*

Theorem 80 [AvE82]. *Suppose S is a Horn clause set and I is a nonempty set of Herbrand models of S, then $\bigcap I$ is also a model of S.*

Proof. Assume that $\bigcap I$ is not a model for S, then $\bigcap I$ falsifies some ground instance $C\theta$ of a clause $C \in S$. If C has a positive literal than $C\theta$ has the form:

$$C\theta = A \vee \neg A_1 \vee \ldots \vee \neg A_m, m \geq 0, \bigcap I \not\models A, \text{and for all } 0 \leq i \leq m, \bigcap I \models A_i.$$

Thus clearly there is a $I_j \in I$ such that $I_j \not\models C\theta$. A similar result can be derived if C has only negative literals. And one can show that this I_j is not a model for S. This contradicts the given facts so the theorem is proved. □

Recall that decision procedures are mainly used to discard partial hyper-links $C_1\alpha \cup C_2\alpha$ (see Definition 53 on Page 51) which have valid $C_2\alpha$. Thus the implication of this theorem is twofold: First, decision procedures for structures of Horn clause sets can be combined directly in such way that a non-ground clause C is valid in $\bigcap I$ if C is valid in all $\bigcap I$. And, such validity is guaranteed. Second, even for non-Horn clause sets, the partial hyper-link $C_1\alpha \cup C_2\alpha$ might have a Horn $C_2\alpha$ for which the validity can be determined also.

The point is that semantic hyper-linking is still complete as long as the discarded partial hyper-link $C_1\alpha \cup C_2\alpha$ has a valid $C_2\alpha$ in $\bigcap I$; namely only "useless" hyper-links are thrown away and this is safe. Decision procedures for decidable structures are used mainly for this purpose. Using multiple semantics is straightforward if only finite semantics are involved. If, however, infinite semantics are used, the decision procedures, if any, can used to determine validity only when Horn clauses are involved.

6.7 Summary

Semantic hyper-linking is independent of the input semantics, so one has freedom to choose any consistent semantics. However, among the infinite number of semantics, we hope to choose one that can help to find the proofs faster. Although it is still not clear what are the exact criteria for a "good" semantics, we now discuss some general guidelines, and try to investigate how different semantics behave in finding proofs.

Recall that the input clauses contain the axioms A and the negated theorem $\neg T$. In general, axioms are satisfiable—they are satisfied by the designated semantics. Since semantic hyper-linking proceeds to find a unsatisfiable set G of ground instances of input clauses, if the axioms are satisfiable, it can be guaranteed that the unsatisfiable ground set contain some ground instances of clauses from $\neg T$. Thus a good semantics should generate such ground instances without much exhaustive enumeration. However, it is difficult to determine beforehand how much exhaustive enumeration one semantics might incur. For example, ground instances can also be generated by the model finding strategies, sometimes one seemingly bad semantics might perform better than a seemingly good semantics because the model finding strategies perform better.

Semantic hyper-linking generates instances of input clauses and, unlike resolution, does not combine literals from different clauses. Also, semantic checks are applied to instances of input clauses or a subset. Thus using semantics with semantic hyper-linking is not as expensive as with resolution style method because less literals are involved in each semantic check. Also, the use of decidable semantics is not essential to the completeness of semantic hyper-linking, so any semantics with evaluation procedures for ground literals, can be used with semantic hyper-linking to find the proofs. Also, semantics often performs well when the needed ground instances are not too big and the search space is of moderate size. This is true especially for non-Horn clauses (such as the EXQ problems).

Chapter 7

Discussions

7.1 Domain Knowledge Vs. Human Control

To prove hard theorems, often theorem provers need either domain-specific knowledge (for example, the variable elimination method for dense linear ordering [BH80, Hin92], or term rewriting techniques) or human control (for example, set of support and use of different resolutions). Quite often proofs cannot be found if neither is used.

Domain-specific knowledge allows focus on *the theorems and the axioms* during search for proofs; human control focuses on the *proof finding process*. It is difficult to see which information is likely to be widely used once theorem proving techniques become powerful enough to serve as a solid foundation of automated reasoning. However, domain knowledge often is readily available with the theorems, while proper human control has to be set by a human and cannot be automated.

Automatically using proper proving strategies is one of the 33 hard research problems proposed by Wos [Wos88]. There have been some attempts to answer this question. Lee and Plaisted [Lee90] devised an expert system to systematically try different control strategies automatically. McCune uses an automated mode in his widely used prover `Otter` [McC94] to determine proper proving strategies to use by examining the input.

Gelernter [Gel59] was among the first to use of domain-specific knowledge in theorem proving. *Semantically guided* theorem proving has been studied since then. Two common approaches in using domain specific knowledge are:

1. Use semantic information to prune the search space. In general the approach is to discard unsuccessful subgoals in backward chaining. Examples include [Gel59, Rei73, San80, Hen76, Sla67, NP90, Wan85, Luc68, SL92].

2. Use special inference rules applicable in the problems domain. Examples include term rewriting in equality [Plaar, DJ90], and Bledsoe and Hines' variable elimination inference rule for dense linear ordering [BH80, Hin92].

Using semantics in automated theorem provers is natural in general since automated reasoning should involve both the problems and the related domain knowledge. Also we think it is probably the better method to conduct efficient and effective proof search since mathematicians rely on semantics to prove hard theorems. We believe that special strategies are probably embodied in semantics when proving hard theorems.

Some results have been shown in the literature but in general semantics has not had positive help to theorem provers. The main reason is the semantic check is often too expensive, especially when used in resolution style theorem provers in which literals from different clauses are combined and resolvents grow larger and larger.

Semantic hyper-linking tries to address this problem by taking the instance-based approach in which only instances of the input clauses are generated so semantic checks are not as expensive as in resolution. Instead of trying to generate empty (false) resolvents, semantic hyper-linking takes a model searching process to look for an unsatisfiable ground instance set.

Also, the model finding strategies, that find logical consequences in contradiction to the models found for the ground set, has a philosophy parallel to what mathematicians do: focus on the derived facts (the ground instance set) and proceed with the help from those facts.

7.2 A Unified View

There are three major methods in our proving techniques—rough resolution to eliminate the large literals, then UR resolution to eliminate the Horn part of a proof, then what is left is non-Horn with small literals and on those kinds of problems semantic hyper-linking is effective. In this section, we will present a unified view of the cooperation of those three methods.

First let us review ordered resolution. This is one refinement of resolution [Lov78, CL73] which uses an ordering on literals in a clause and restricts the application of resolution such that the subset of resolution may only contain clauses that are maximal in the ordering.

A-ordering (atom-ordering) $\preceq_A$ is one such example. Let $\|L\|$ denote the corresponding atom for a given literal L. A-ordering is an ordering which, for a clause set S, orders a subset T of all literal instances (containing only symbols in the alphabet of S) such that for all $\|L_1\|$, $\|L_2\| \in T$ and all substitution θ, α over S,

- $\|L_1\theta\| \in T$;
- $\|L_1\| \prec_A \|L_2\|$ implies $\|L_1\theta\| \prec_A \|L_2\alpha\|$; and
- $\|L_1\theta\| \preceq_A \|L_1\|$.

Suppose S is an unsatisfiable set of clauses, and let T be an unsatisfiable set of ground instances. Let M be the set of all atoms of T of size d or more, for some integer d. Then it is possible to obtain by A-ordering resolution from S, a set S' of clauses, such that S' has an unsatisfiable set T' of ground instances, such that all the atoms in T' are of size d or less. Also, from S', by UR resolution it is possible to obtain a set S'' of clauses such that S'' has an unsatisfiable set T'' of ground instances such that T'' has only atoms of size d or less, and such that T'' contains no unit clauses. This set T'' is a set that is likely to be good for semantic hyper-linking since the literals are small and there are no unit clauses (this means that the Horn part has been removed).

As stated, this is actually trivial because we can just let T' be the set containing the empty clause, since A-ordering is complete. So we can add some substance to this result as follows: Suppose T has n atoms of size greater than d. Then we can say that in the obtaining of S' from S, only n rounds of A-ordering resolution are needed. So for example if $n = 1$ then only one round is needed. Also, if each atom of size greater than d appears only once positively and once negatively in T, then we can show that a total of only n A-ordering resolutions are needed to obtain S'

from S. Then, at most (number of literals in T) unit resolutions on S' are needed to remove all the unit literals and obtain S''; this is because the A-ordering resolutions will not increase the number of occurrences of any literal in going from T to T'. If each atom of size greater than d appears m times positively and m times negatively, then it may require m^2 A-ordering resolutions to eliminate it; this can increase the number of occurrences of other literals by a factor of m. Thus the next time the literal to resolve on may appear m^2 times positively and m^2 times negatively, so we may need to do m^4 A-ordering resolutions to eliminate it. This may cause the number of occurrences of other literals to multiply by a factor of m^2, so the next literal to resolve away may appear m^4 times. We get a sequence of $m, m^2, m^4 \ldots m^{2^d}$ or so, so the number of resolutions needed to get rid of all these large literals can be as much as $m^{(1+2+\ldots+2^d)}$ or $m^{(2^{(d+1)}-1)}$, approximately m^{2^d} which is double exponential in d. If d is very small, say one or two, this is not too bad. And, in any case, we can say that the number of A-ordering resolutions needed is never more than $2^{(number\ of\ atoms\ in\ T)}$, since one can prove the empty clause in this many resolutions.

Such a unified view is also a useful device to understand how we can get proofs. When we look at sets of ground instances for various proofs, we can apply these methods and see which literals we should be getting rid of by rough resolution and find out, if not, why not, and if we could get rid of certain literals, could we get the proof. It provides us an excellent tool to understand how we could in theory get any proof and what it is that makes some proofs too hard for our prover.

It is also interesting that A-ordering resolution is a decision procedure for certain clause sets. So if we did A-ordering resolution instead of rough resolution, we could decide those clause sets too when we found that there were no more A-ordering resolutions to do.

Chapter 8

Implementation and Results

8.1 The Implementation

We have implemented a prover, called `CLIN-S`, in Prolog based on our method. There are three major components in `CLIN-S`: semantic hyper-linking, rough resolution and UR resolution. Rough resolution eliminates big literals, UR resolution eliminates Horn parts of proofs, and what is left is non Horn and with small literals, where semantic hyper linking performs well. Work among different components is well balanced by the time spent on them. We will now discuss some implementation details of various methods discussed earlier.

8.1.1 Semantic Hyper-Linking

In `CLIN-S` clauses are represented as lists of literals. Recall the discussion in Section 3.3. During the early stage of semantic hyper-linking, given a clause set S and a set of eligible literals, `CLIN-S` tries to first generate all the partial hyper-links from S and then generate the smallest ground instances from those partial hyper-links. Both processes use the hyper-linking operation.

Partial Hyper-Links Suppose C is expressed as $C_1 \cup C_2$ and $C_1 \cap C_2 = \phi$. Given a clause C and a set of eligible literals, the partial hyper-links are generated by checking all the possible subsets C_1 of C in which all literals can be hyper-linked with the eligible literals. Let the resulting substitution be α. The partial hyper-link $C\alpha$ is kept if $C_2\alpha$ is not valid ($C_2\alpha$ might have a ground instance in which all literals are false). By checking $C_2\alpha$, the number of partial hyper-links can be greatly reduced.

Ground Instance Generation Given all the partial hyper-links generated by the previous process, minimal ground instances are generated in this process. Recall that we use the Herbrand base enumeration of the literals in $C_2\alpha$. To make the enumeration efficient, we do not examine those $C_2\alpha$ one by one. Instead, we collect all the non-ground literals (with duplicates removed) from all the $C_2\alpha$, and do Herbrand base enumeration on them. We generate all possible true ground literals—starting from size 1—from those literals. At each size, all $C_2\alpha$ are hyper-linked with the enumerated ground literals. As soon as some $C_2\alpha$ are fully linked, we have generated ground instances (since $C_1\alpha$ are already ground)..

To assure that minimal ground instances are generated, the sizes of $C_1\alpha$ need to be considered when hyper-linking $C_2\alpha$. This is straightforward.

Consider the $C_2\alpha = \{\, p(f(X), g(Y), a), q(X,Y)\,\}$. Obviously it is not necessary to enumerate $q(X,Y)$ until at least size 6 (the size of $p(f(X), g(Y), a)$). If $q(X,Y)$ does not appear in any other $C_2\alpha$, one can do without $q(X,Y)$ in early stages of the enumeration. Also, in this particular example, once $p(f(X), g(Y), a)$ is ground, $q(X,Y)$ becomes ground as well. So one need enumerate only the former literal and do not use the latter at all. Such optimizations can make the hyper-linking operation more efficient.

8.1.2 The Sup-Inf procedure

The Sup-Inf procedure is implemented as the only built-in decision procedure (for semantics in Bledsoe real arithmetic). The implementation of `CLIN-S` is based on the procedure discussed in [Bun83].

Piecewise linear functions such as $max(x,y)$ or $abs(x)$ commonly arise in natural semantics of Bledsoe real arithmetic. Their presence requires expanding the corresponding function definitions before applying the Sup-Inf procedure. Such definition expansion can be done in two ways: One way is to expand the innermost functions, each time replacing the functions by definitions in DNF conjuncted with the conditions. For example: to expand definitions of max and abs in the formula

$$max(a,b) + abs(c) < d$$

we can obtain

$$\begin{aligned}
&(a + c < d \wedge \underbrace{b \leq a \wedge 0 \leq c}_{condition}) \vee (a - c < d \wedge \underbrace{b \leq a \wedge c < 0}_{condition}) \\
&\vee(b + c < d \wedge \underbrace{a < b \wedge 0 \leq c}_{condition}) \vee (b - c < d \wedge \underbrace{a < b \wedge c < 0}_{condition})
\end{aligned}$$

According to the experimental results, this approach is not efficient probably due to the complicated formulas after the definitions are expanded, that are particularly difficult for DNF transformation.

We use another approach which replaces each piecewise linear function by a new variable and add the functions definitions at the outer level. For example, we can define $max(x,y)$ as

$$max(x,y) = \begin{cases} x & \text{if } y \leq x \\ y & \text{otherwise} \end{cases}$$

In first-order formula, we can represent $max(x,y)$ as

$$((y \leq x) \supset max(x,y) = x) \wedge ((x < y) \supset max(x,y) = y)$$

When the function $max(x,y)$ in a formula F is expanded, it is first replaced by a new variable, say v. Suppose the formula after the variable replacement is F'. The definition for $max(x,y)$ is added as a condition to form the expanded formula

$$\underbrace{(((y \leq x) \supset v = x) \wedge ((x < y) \supset v = y))}_{definition\ \ for\ \ v=max(x,y)} \supset F'$$

In general for a formula F, we replace piecewise functions by distinct new variables to get F' and then place the conjunction of the definitions as the condition for F':

$$(\bigwedge \textit{piecewise function definitions}) \supset F'$$

The new variables are used in the corresponding function definitions. For example, consider the formula

$$F : max(a, b) + abs(c) < d$$

We can replace $max(a, b)$ by a new variable $v1$, $abs(c)$ by $v2$. We then expand F to obtain the following formula:

$$[\underbrace{(b \leq a \supset v1 = a) \wedge (a < b \supset v1 = b)}_{definition\ for\ v1=max(a,b)} \wedge \underbrace{(0 \leq c \supset v2 = c) \wedge (c < 0 \supset v2 = -c)}_{definition\ for\ v2=abs(c)}]$$
$$\supset \underbrace{(v1 + v2 < d)}_{F'}$$

The expanded formula can then be checked using the Sup-Inf procedure.

The Sup-Inf procedure demonstrates the power of semantic hyper-linking with a decidable semantics. Other decision procedures for decidable semantics can be implemented similarly. In fact the prover has been implemented in such a way that new decision procedures can be added by the user using Prolog procedures although this has not been officially a feature of our prover.

8.2 Balancing Different Components

We use two criterion to control the collaboration among different components in the prover:

1. Evenly spending time on various prover components. The idea is to use time spent on one major component in a semantic hyper-linking round, as a time bound on other components in the same round. This has some disadvantage since different techniques generate instances at different rates. Thus we might have more instances generated by one technique than others.

   For example, ground units generated by UR resolution are put in the ground model. If there are many of them, we might be involved in more forward chaining instead of backward chaining which is the essential concept of semantic hyper-linking.

We can also restrict how many instances can be passed between components.

2. Control the number of instances generated by each component. We might want to favor some components over others; those favored should generate more instances.

8.3 Experimental Results

We have proved hard theorems like IMV, AM8, EXQ1, EXQ2, EXQ3, and others. Those problems are hard enough that other provers have to use special strategies or built-in inference rules to obtain the proofs. We submitted the theorems with their semantics and the prover found the proofs without other human intervention.

With no guidance, `CLIN-S` has proved several hard theorems including problems in set theory, temporal logic, modal logic, geometry, standard toy problems, and the propositional calculus. Semantics sometimes helps to solve those hard problems. Table 8.1 contains results on some hard theorems using `CLIN-S` compared with `Otter` and `CLIN`. All the results, measured in seconds except when noted, are obtained on a DEC 5500. Note that those results mainly show that semantic hyper-linking is powerful and could prove those hard theorems without special guidance.

IMV [BB82] (intermediate value theorem) and AM8 [Ble83] (attaining maximum theorem) are two very hard theorems in real analysis. LIM1–3 are the first three LIM+ problems proposed by Bledsoe [Ble90].

EXQ problems, EXQ1–3, are three examples from quantification theory [Wan65] in the order of increasing difficulty. They are difficult for most powerful theorem provers probably due to the non-Horn property. We proved all three EXQ problems on the first try fairly quickly. We think EXQ3 has never been proven before fully automatically.

HALT is the undecidability of the halting problem [Bru91]. We use the version from non-standard clause translation [PG86]. Since it is difficult to devise reasonable semantics for the halting problem, we used a trivial semantics in which the true literals

Problems	CLIN-S (in Prolog)		Otter (in C)		CLIN* (in Prolog)
	Natural Semantics	Trivial Semantics	No† Guidance	Better‡ Guidance	
IMV	1,600	$> 3,000$	> 3 hours	> 3 hours	> 3 hours¶
AM8	973	1,474	> 3 hours	> 3 hours	> 3 hours¶
HALT (nonstand)	—§	129	> 3 hours	> 3 hours	10,476
LIM1	99	464	> 3 hours	> 3 hours	> 3 hours
LIM2	81	507	> 3 hours	> 3 hours	> 3 hours
LIM3	2,265	$> 5,000$	> 3 hours	> 3 hours	> 3 hours
EXQ1	211	105	> 3 hours	> 3 hours	> 3 hours
EXQ2	94	90	> 3 hours	> 3 hours	> 3 hours
EXQ3	150	86	> 3 hours	> 3 hours	> 3 hours
P38 (stand)	—§	1,374	> 3 hours	> 3 hours	> 3 hours
P38 (nonstand)	—§	58	> 3 hours	> 3 hours	> 3 hours

*Use default setting
†Use hyper-resolution and UR resolution with every clause in the set of support
‡Use binary, UR and hyper-resolution with the negated theorem in the set of support
¶Can be proved in < 100 sec using predicate replacement
§Not attempted

Table 8.1: Performance of semantic hyper-linking on some hard theorems

are exactly those positive ones. Even with such a bad choice for semantics, our prover still can prove it easily. P38 is the 38th problem in [Pel86]. Again, a natural semantics is not available and we tried `CLIN-S` on both versions from standard and non-standard clause translation.

In the following three tables, we will list timing results by `CLIN-S` running on some problems including those in the set theory, the temporal logic, geometry and the group theory.

Table 8.2 shows the results by using different semantics. Time is measured in seconds; "Rnd" stands for the number of rounds to get the proofs; "Type" of proof signifies how proofs are obtained:

"—" Proof not found (exceeding time bound),

HM Hyper-Linking: a contradiction is found during hyper-linking (including unit conflicts),

MF Model finding: an unsatisfiable ground set is obtained,

RR Rough resolution: an empty clause is obtained during rough resolution,

US Unit simplification: found by `CLIN` unit simplification procedure,

UR UR resolution: a unit conflict is found during UR resolution.

"Cls" denotes the number of clauses retained after a proof is found. Note that this does not include those clauses that are already generated but not yet used.

"No" of models stands for the number of models being used (see Section 6.6). "Type" of models is one of the following:

FIN Finite natural semantics,

INF Infinite natural semantics,

P Trivial semantics in which all atoms in the Herbrand base are true,

N Trivial semantics in which all atoms in the Herbrand base are false.

Table 8.3 shows the different components used to prove the problems. "Flags" stand for major components in `CLIN-S`:

FL Model filtering,

NP Natural replacement,

TR Simple term rewriting,

MP Minimal replacement,

RR Rough resolution,

UR UR resolution.

Table 8.4 shows the model contradictions found by each components.

Table 8.2: The effects of natural and trivial semantics

No.	Problem	Proof				Model	
		Time	Rnd	Type	Cls	No	Type
1	SYN036-4	102.6	6	MF	104	1	P
2	SYN038-1	800.3	12	MF	570	1	N
3	SYN038-1	195.5	13	MF	311	1	P
4	SYN086-1	75.9	1	MF	68	1	P
5	SYN086-2	15.5	1	HM	57	1	P
6	am8	2532.1	2	HM	169	1	INF
7	andrew_nonstd	54.8	1	MF	72	1	P
8	bledsoe1	53.6	2	HM	44	1	INF
9	bledsoe2	61.5	2	HM	57	1	INF
10	bledsoe3	1527.1	4	HM	362	1	INF
11	ex4t1	95.0	3	MF	65	1	N
12	ex4t2	94.1	3	MF	64	1	N
13	ex6t1	181.7	4	UR	68	1	N
14	ex6t1	383.3	4	UR	64	1	P
15	ex6t2	189.8	5	UR	148	1	N
16	ex6t2	290.7	4	UR	60	1	P
17	halting_nonstd	126.2	1	HM	277	1	P
18	imv	357.7	5	MF	143	1	INF
19	ipc_i1	1.5	1	UR	12	1	FIN
20	ipc_ip1	3383.5	6	UR	105	1	FIN
21	ipc_p1	1356.4	4	UR	77	1	FIN
22	ipc_s1	56.5	1	UR	21	1	FIN
23	ls37	245.3	1	UR	103	2	FIN
24	p38.nonstand	58.6	1	MF	120	1	N
25	p38.stand	1122.9	1	HM	225	1	N
26	sam	208.1	1	HM	233	1	FIN
27	set10	229.6	1	HM	103	1	P
28	set10	341.9	1	HM	106	2	INF
29	set11	109.5	1	HM	102	1	P
30	set11	297.9	1	HM	105	2	INF
31	set12	57.6	1	HM	104	1	P
32	set12	188.7	1	HM	107	2	INF
33	set13	277.1	1	HM	104	1	P
34	set13	186.4	2	HM	109	2	INF

(continued on next page)

Table 8.2: The effects of natural and trivial semantics (continued)

No.	Problem	Proof				Model	
		Time	Rnd	Type	Cls	No	Type
35	set14	272.6	1	HM	105	1	P
36	set14	389.1	1	HM	108	2	INF
37	set15	228.4	1	HM	79	1	P
38	set15	513.4	1	HM	83	2	INF
39	set16	58.5	1	HM	106	1	P
40	set16	172.6	1	HM	106	2	INF
41	set17	175.8	1	HM	104	1	P
42	set17	577.6	2	HM	263	2	INF
43	set18	277.3	1	HM	103	1	P
44	set18	633.5	2	HM	178	2	INF
45	set19	335.9	1	HM	104	1	P
46	set19	704.8	2	HM	175	2	INF
47	set1	183.1	1	HM	103	1	P
48	set1	401.8	1	HM	107	2	INF
49	set20	226.8	1	HM	104	1	P
50	set21	297.1	1	HM	104	1	P
51	set21	419.9	1	HM	108	2	INF
52	set22	295.3	1	HM	105	1	P
53	set22	416.2	1	HM	108	2	INF
54	set23	0.6	0	SP	43	1	P
55	set23	0.6	0	SP	43	2	INF
56	set24	55.5	1	HM	98	1	P
57	set24	142.7	1	HM	104	2	INF
58	set25	0.6	0	SP	45	1	P
59	set25	0.6	0	SP	45	2	INF
60	set26	318.2	1	HM	105	1	P
61	set27	55.7	1	HM	108	1	P
62	set27	19.3	1	HM	51	2	INF
63	set28	56.0	1	HM	108	1	P
64	set28	31.4	1	HM	55	2	INF
65	set29	2.5	1	HM	53	1	P
66	set29	33.4	1	HM	59	2	INF
67	set2	54.9	1	HM	105	1	P

(continued on next page)

Table 8.2: The effects of natural and trivial semantics (continued)

No.	Problem	Proof				Model	
		Time	Rnd	Type	Cls	No	Type
68	set2	178.1	1	HM	109	2	INF
69	set30	176.2	1	HM	104	1	P
70	set30	570.6	2	HM	263	2	INF
71	set35	56.1	1	HM	109	1	P
72	set35	182.8	1	HM	112	2	INF
73	set36	0.1	0	US	45	1	P
74	set36	0.1	0	US	45	2	INF
75	set3	57.2	1	HM	102	1	P
76	set3	184.7	1	HM	107	2	INF
77	set40	208.6	1	HM	103	1	P
78	set40	556.2	2	HM	155	2	INF
79	set41	54.3	1	HM	126	1	P
80	set41	111.0	1	HM	128	2	INF
81	set4	57.5	1	HM	105	1	P
82	set4	183.8	1	HM	107	2	INF
83	set5	235.5	1	HM	104	1	P
84	set5	630.6	2	HM	161	2	INF
85	set6	331.3	1	HM	104	1	P
86	set6	624.3	2	HM	160	2	INF
87	set7	324.9	1	HM	103	1	P
88	set7	738.0	2	HM	154	2	INF
89	set8	234.0	1	HM	103	1	P
90	set8	548.8	2	HM	155	2	INF
91	set9	230.8	1	HM	103	1	P
92	set9	637.8	2	HM	162	2	INF
93	t10	330.1	1	HM	170	1	P
94	t11	792.5	1	HM	157	1	P
95	t11	14540.7	2	—	444	1	INF
96	t13	158.8	1	HM	174	1	P
97	t13	27499.5	1	HM	2422	1	INF
98	t1	1867.8	1	UR	317	1	P
99	t1	2337.9	1	UR	246	1	INF
100	t2	77.1	1	HM	278	1	P

(continued on next page)

Table 8.2: The effects of natural and trivial semantics (continued)

No.	Problem	Proof				Model	
		Time	Rnd	Type	Cls	No	Type
101	t2	912.3	1	HM	215	1	INF
102	t3	78.0	1	HM	289	1	P
103	t3	914.6	1	HM	236	1	INF
104	t6	11018.4	3	—	561	1	P
105	t6	2015.7	1	HM	206	1	INF
106	t7	4659.8	1	HM	232	1	INF
107	tl10	240.9	2	HM	229	1	P
108	tl10	358.4	2	HM	305	1	INF
109	tl11	240.4	2	HM	229	1	P
110	tl11	358.1	2	HM	305	1	INF
111	tl12	253.8	2	HM	241	1	P
112	tl12	310.2	1	MF	125	1	INF
113	tl13	255.4	2	HM	239	1	P
114	tl13	448.2	1	MF	124	1	INF
115	tl14	253.6	2	HM	249	1	P
116	tl14	444.4	1	HM	126	1	INF
117	tl15	518.9	2	MF	314	1	P
118	tl15	378.2	1	MF	152	1	INF
119	tl16	300.8	2	HM	205	1	P
120	tl16	381.4	1	HM	101	1	INF
121	tl17	305.6	2	HM	206	1	P
122	tl17	329.5	1	HM	101	1	INF
123	tl18	1834.1	2	HM	199	1	INF
124	tl19	219.4	2	HM	199	1	P
125	tl19	246.8	1	HM	103	1	INF
126	tl1	149.6	2	HM	197	1	P
127	tl1	265.5	2	HM	165	1	INF
128	tl23	194.7	2	HM	203	1	P
129	tl23	200.9	1	MF	111	1	INF
130	tl24	489.6	2	HM	346	1	P
131	tl24	397.4	2	HM	305	1	INF
132	tl25	2836.9	2	—	382	1	INF
133	tl26	582.5	2	HM	355	1	P

(continued on next page)

Table 8.2: The effects of natural and trivial semantics (continued)

No.	Problem	Proof				Model	
		Time	Rnd	Type	Cls	No	Type
134	tl26	305.9	1	HM	115	1	INF
135	tl27	862.3	2	HM	441	1	P
136	tl27	522.7	2	HM	472	1	INF
137	tl2	171.3	2	HM	197	1	P
138	tl2	269.8	2	HM	164	1	INF
139	tl30	637.4	2	HM	316	1	P
140	tl30	646.8	1	HM	144	1	INF
141	tl31	269.9	2	HM	229	1	P
142	tl31	394.9	2	HM	305	1	INF
143	tl32	408.9	2	HM	285	1	P
144	tl32	525.2	2	HM	472	1	INF
145	tl33	404.7	2	HM	285	1	P
146	tl33	524.0	2	HM	472	1	INF
147	tl34	885.0	2	HM	441	1	P
148	tl34	544.6	2	HM	472	1	INF
149	tl35	405.3	2	HM	285	1	P
150	tl35	522.0	2	HM	472	1	INF
151	tl37	972.6	2	HM	441	1	P
152	tl37	628.8	2	HM	472	1	INF
153	tl38	456.1	2	HM	285	1	P
154	tl38	591.1	2	HM	472	1	INF
155	tl3	255.2	2	HM	199	1	P
156	tl3	281.6	1	HM	103	1	INF
157	tl4	190.6	2	HM	199	1	P
158	tl4	221.2	1	HM	103	1	INF
159	tl5	181.8	2	HM	202	1	P
160	tl5	197.0	1	MF	110	1	INF
161	tl6	239.1	2	HM	229	1	P
162	tl6	356.3	2	HM	305	1	INF
163	tl7	485.8	2	HM	354	1	P
164	tl7	297.1	1	HM	115	1	INF
165	tl8	272.4	2	HM	238	1	P
166	tl8	235.5	1	HM	115	1	INF

(continued on next page)

Table 8.2: The effects of natural and trivial semantics (continued)

No.	Problem	Proof				Model	
		Time	Rnd	Type	Cls	No	Type
167	tl9	237.4	2	HM	229	1	P
168	tl9	356.6	2	HM	299	1	INF
169	wos15	250.4	1	UR	206	1	INF
170	wos19	285.2	1	HM	221	2	FIN
171	wos21	325.1	1	UR	149	1	FIN
172	wos31_EXQ1	97.5	1	HM	44	1	INF
173	wos31_EXQ2	119.9	1	MF	70	1	INF
174	wos31_EXQ3	90.9	1	MF	73	1	INF

Table 8.3: The effects of using different components

No.	Problem	Proof		Model		Flags					
						F	N	T	M	R	U
		Time	Type	No	Type	L	P	R	P	R	R
1	SYN036-4	102.6	MF	1	P	•	•		•	•	•
2	SYN038-1	800.3	MF	1	N	•	•		•	•	•
3	SYN038-1	195.5	MF	1	P	•	•		•	•	•
4	SYN086-1	75.9	MF	1	P	•	•		•	•	•
5	SYN086-2	15.5	HM	1	P	•	•		•	•	•
6	am8	2532.1	HM	1	INF	•	•		•	•	•
7	andrew_nonstd	54.8	MF	1	P	•	•		•	•	•
8	bledsoe1	53.6	HM	1	INF	•	•		•	•	•
9	bledsoe2	61.5	HM	1	INF	•	•		•	•	•
10	bledsoe3	1527.1	HM	1	INF	•	•			•	•
11	ex4t1	95.0	MF	1	N	•	•		•	•	•
12	ex4t2	94.1	MF	1	N	•	•		•	•	•
13	ex6t1	181.7	UR	1	N	•	•		•	•	•
14	ex6t1	383.3	UR	1	P	•	•		•	•	•
15	ex6t2	189.8	UR	1	N	•	•		•	•	•
16	ex6t2	290.7	UR	1	P	•	•		•	•	•
17	halting_nonstd	126.2	HM	1	P	•	•		•	•	•
18	imv	357.7	MF	1	INF	•	•			•	
19	ipc_i1	1.5	UR	1	FIN	•	•		•	•	•
20	ipc_ip1	3383.5	UR	1	FIN	•	•		•	•	•
21	ipc_p1	1356.4	UR	1	FIN	•	•		•	•	•
22	ipc_s1	56.5	UR	1	FIN	•	•		•	•	•
23	ls37	245.3	UR	2	FIN	•	•	•	•	•	•
24	p38.nonstand	58.6	MF	1	N	•	•			•	•
25	p38.stand	1122.9	HM	1	N	•	•		•	•	•
26	sam	208.1	HM	1	FIN	•	•	•	•	•	•
27	set10	229.6	HM	1	P	•	•		•	•	•
28	set10	341.9	HM	2	INF	•	•		•	•	•
29	set11	109.5	HM	1	P	•	•		•	•	•
30	set11	297.9	HM	2	INF	•	•		•	•	•
31	set12	57.6	HM	1	P	•	•		•	•	•
32	set12	188.7	HM	2	INF	•	•		•	•	•
33	set13	277.1	HM	1	P	•	•		•	•	•
34	set13	186.4	HM	2	INF	•	•		•	•	•

(continued on next page)

Table 8.3: The effects of using different components (continued)

No.	Problem	Proof		Model		Flags					
						F	N	T	M	R	U
		Time	Type	No	Type	L	P	R	P	R	R
35	set14	272.6	HM	1	P	•	•		•	•	•
36	set14	389.1	HM	2	INF	•	•		•	•	•
37	set15	228.4	HM	1	P	•	•		•	•	•
38	set15	513.4	HM	2	INF	•	•		•	•	•
39	set16	58.5	HM	1	P	•	•		•	•	•
40	set16	172.6	HM	2	INF	•	•		•	•	•
41	set17	175.8	HM	1	P	•	•		•	•	•
42	set17	577.6	HM	2	INF	•	•		•	•	•
43	set18	277.3	HM	1	P	•	•		•	•	•
44	set18	633.5	HM	2	INF	•	•		•	•	•
45	set19	335.9	HM	1	P	•	•		•	•	•
46	set19	704.8	HM	2	INF	•	•		•	•	•
47	set1	183.1	HM	1	P	•	•		•	•	•
48	set1	401.8	HM	2	INF	•	•		•	•	•
49	set20	226.8	HM	1	P	•	•		•	•	•
50	set21	297.1	HM	1	P	•	•		•	•	•
51	set21	419.9	HM	2	INF	•	•		•	•	•
52	set22	295.3	HM	1	P	•	•		•	•	•
53	set22	416.2	HM	2	INF	•	•		•	•	•
54	set23	0.6	SP	1	P	•	•		•	•	•
55	set23	0.6	SP	2	INF	•	•		•	•	•
56	set24	55.5	HM	1	P	•	•		•	•	•
57	set24	142.7	HM	2	INF	•	•		•	•	•
58	set25	0.6	SP	1	P	•	•		•	•	•
59	set25	0.6	SP	2	INF	•	•		•	•	•
60	set26	318.2	HM	1	P	•	•		•	•	•
61	set27	55.7	HM	1	P	•	•		•	•	•
62	set27	19.3	HM	2	INF	•	•		•	•	•
63	set28	56.0	HM	1	P	•	•		•	•	•
64	set28	31.4	HM	2	INF	•	•		•	•	•
65	set29	2.5	HM	1	P	•	•		•	•	•
66	set29	33.4	HM	2	INF	•	•		•	•	•
67	set2	54.9	HM	1	P	•	•		•	•	•

(continued on next page)

Table 8.3: The effects of using different components (continued)

No.	Problem	Proof		Model		Flags					
						F	N	T	M	R	U
		Time	Type	No	Type	L	P	R	P	R	R
68	set2	178.1	HM	2	INF	•	•		•	•	•
69	set30	176.2	HM	1	P	•	•		•	•	•
70	set30	570.6	HM	2	INF	•	•		•	•	•
71	set35	56.1	HM	1	P	•	•		•	•	•
72	set35	182.8	HM	2	INF	•	•		•	•	•
73	set36	0.1	US	1	P	•	•		•	•	•
74	set36	0.1	US	2	INF	•	•		•	•	•
75	set3	57.2	HM	1	P	•	•		•	•	•
76	set3	184.7	HM	2	INF	•	•		•	•	•
77	set40	208.6	HM	1	P	•	•		•	•	•
78	set40	556.2	HM	2	INF	•	•		•	•	•
79	set41	54.3	HM	1	P	•	•		•	•	•
80	set41	111.0	HM	2	INF	•	•		•	•	•
81	set4	57.5	HM	1	P	•	•		•	•	•
82	set4	183.8	HM	2	INF	•	•		•	•	•
83	set5	235.5	HM	1	P	•	•		•	•	•
84	set5	630.6	HM	2	INF	•	•		•	•	•
85	set6	331.3	HM	1	P	•	•		•	•	•
86	set6	624.3	HM	2	INF	•	•		•	•	•
87	set7	324.9	HM	1	P	•	•		•	•	•
88	set7	738.0	HM	2	INF	•	•		•	•	•
89	set8	234.0	HM	1	P	•	•		•	•	•
90	set8	548.8	HM	2	INF	•	•		•	•	•
91	set9	230.8	HM	1	P	•	•		•	•	•
92	set9	637.8	HM	2	INF	•	•		•	•	•
93	t10	330.1	HM	1	P	•	•		•	•	•
94	t11	792.5	HM	1	P	•	•		•	•	•
95	t11	14540.7	—	1	INF	•	•		•	•	•
96	t13	158.8	HM	1	P	•	•		•	•	•
97	t13	27499.5	HM	1	INF	•	•		•	•	•
98	t1	1867.8	UR	1	P	•	•		•	•	•
99	t1	2337.9	UR	1	INF	•	•		•	•	•
100	t2	77.1	HM	1	P	•	•		•	•	•

(continued on next page)

Table 8.3: The effects of using different components (continued)

No.	Problem	Proof		Model		Flags					
						F	N	T	M	R	U
		Time	Type	No	Type	L	P	R	P	R	R
101	t2	912.3	HM	1	INF	•	•		•	•	•
102	t3	78.0	HM	1	P	•	•		•	•	•
103	t3	914.6	HM	1	INF	•	•		•	•	•
104	t6	11018.4	—	1	P	•	•		•	•	•
105	t6	2015.7	HM	1	INF	•	•		•	•	•
106	t7	4659.8	HM	1	INF	•	•		•	•	•
107	tl10	240.9	HM	1	P	•	•		•	•	•
108	tl10	358.4	HM	1	INF	•	•		•	•	•
109	tl11	240.4	HM	1	P	•	•		•	•	•
110	tl11	358.1	HM	1	INF	•	•		•	•	•
111	tl12	253.8	HM	1	P	•	•		•	•	•
112	tl12	310.2	MF	1	INF	•	•		•	•	•
113	tl13	255.4	HM	1	P	•	•		•	•	•
114	tl13	448.2	MF	1	INF	•	•		•	•	•
115	tl14	253.6	HM	1	P	•	•		•	•	•
116	tl14	444.4	HM	1	INF	•	•		•	•	•
117	tl15	518.9	MF	1	P	•	•		•	•	•
118	tl15	378.2	MF	1	INF	•	•		•	•	•
119	tl16	300.8	HM	1	P	•	•		•	•	•
120	tl16	381.4	HM	1	INF	•	•		•	•	•
121	tl17	305.6	HM	1	P	•	•		•	•	•
122	tl17	329.5	HM	1	INF	•	•		•	•	•
123	tl18	1834.1	HM	1	INF	•	•		•	•	•
124	tl19	219.4	HM	1	P	•	•		•	•	•
125	tl19	246.8	HM	1	INF	•	•		•	•	•
126	tl1	149.6	HM	1	P	•	•		•	•	•
127	tl1	265.5	HM	1	INF	•	•		•	•	•
128	tl23	194.7	HM	1	P	•	•		•	•	•
129	tl23	200.9	MF	1	INF	•	•		•	•	•
130	tl24	489.6	HM	1	P	•	•		•	•	•
131	tl24	397.4	HM	1	INF	•	•		•	•	•
132	tl25	2836.9	—	1	INF	•	•		•	•	•
133	tl26	582.5	HM	1	P	•	•		•	•	•

(continued on next page)

Table 8.3: The effects of using different components (continued)

No.	Problem	Proof		Model		Flags					
						F	N	T	M	R	U
		Time	Type	No	Type	L	P	R	P	R	R
134	tl26	305.9	HM	1	INF	•	•		•	•	•
135	tl27	862.3	HM	1	P	•	•		•	•	•
136	tl27	522.7	HM	1	INF	•	•		•	•	•
137	tl2	171.3	HM	1	P	•	•		•	•	•
138	tl2	269.8	HM	1	INF	•	•		•	•	•
139	tl30	637.4	HM	1	P	•	•		•	•	•
140	tl30	646.8	HM	1	INF	•	•		•	•	•
141	tl31	269.9	HM	1	P	•	•		•	•	•
142	tl31	394.9	HM	1	INF	•	•		•	•	•
143	tl32	408.9	HM	1	P	•	•		•	•	•
144	tl32	525.2	HM	1	INF	•	•		•	•	•
145	tl33	404.7	HM	1	P	•	•		•	•	•
146	tl33	524.0	HM	1	INF	•	•		•	•	•
147	tl34	885.0	HM	1	P	•	•		•	•	•
148	tl34	544.6	HM	1	INF	•	•		•	•	•
149	tl35	405.3	HM	1	P	•	•		•	•	•
150	tl35	522.0	HM	1	INF	•	•		•	•	•
151	tl37	972.6	HM	1	P	•	•		•	•	•
152	tl37	628.8	HM	1	INF	•	•		•	•	•
153	tl38	456.1	HM	1	P	•	•		•	•	•
154	tl38	591.1	HM	1	INF	•	•		•	•	•
155	tl3	255.2	HM	1	P	•	•		•	•	•
156	tl3	281.6	HM	1	INF	•	•		•	•	•
157	tl4	190.6	HM	1	P	•	•		•	•	•
158	tl4	221.2	HM	1	INF	•	•		•	•	•
159	tl5	181.8	HM	1	P	•	•		•	•	•
160	tl5	197.0	MF	1	INF	•	•		•	•	•
161	tl6	239.1	HM	1	P	•	•		•	•	•
162	tl6	356.3	HM	1	INF	•	•		•	•	•
163	tl7	485.8	HM	1	P	•	•		•	•	•
164	tl7	297.1	HM	1	INF	•	•		•	•	•
165	tl8	272.4	HM	1	P	•	•		•	•	•
166	tl8	235.5	HM	1	INF	•	•		•	•	•

(continued on next page)

Table 8.3: The effects of using different components (continued)

No.	Problem	Proof		Model		Flags					
						F	N	T	M	R	U
		Time	Type	No	Type	L	P	R	P	R	R
167	tl9	237.4	HM	1	P	•	•		•	•	•
168	tl9	356.6	HM	1	INF	•	•		•	•	•
169	wos15	250.4	UR	1	INF	•	•		•	•	•
170	wos19	285.2	HM	2	FIN	•	•	•	•	•	•
171	wos21	325.1	UR	1	FIN	•	•		•	•	•
172	wos31_EXQ1	97.5	HM	1	INF	•	•	•	•	•	•
173	wos31_EXQ2	119.9	MF	1	INF	•	•	•	•	•	•
174	wos31_EXQ3	90.9	MF	1	INF	•	•	•	•	•	•

Table 8.4: Model contradictions by each components

No.	Problem	Proof		Model		Model Contradictions				
						F	N	T	M	U
		Time	Type	No	Type	L	P	R	P	R
1	SYN036-4	102.6	MF	1	P	26	7	0	0	0
2	SYN038-1	800.3	MF	1	N	133	61	0	0	0
3	SYN038-1	195.5	MF	1	P	12	13	0	0	0
4	SYN086-1	75.9	MF	1	P	2	6	0	0	0
5	SYN086-2	15.5	HM	1	P	0	0	0	1	0
6	am8	2532.1	HM	1	INF	0	0	0	1	0
7	andrew_nonstd	54.8	MF	1	P	0	0	0	0	0
8	bledsoe1	53.6	HM	1	INF	1	0	0	0	0
9	bledsoe2	61.5	HM	1	INF	1	0	0	0	0
10	bledsoe3	1527.1	HM	1	INF	1	0	0	0	0
11	ex4t1	95.0	MF	1	N	9	0	0	0	0
12	ex4t2	94.1	MF	1	N	10	0	0	0	0
13	ex6t1	181.7	UR	1	N	0	0	0	0	0
14	ex6t1	383.3	UR	1	P	2	2	0	0	0
15	ex6t2	189.8	UR	1	N	0	0	0	0	0
16	ex6t2	290.7	UR	1	P	1	1	0	0	0
17	halting_nonstd	126.2	HM	1	P	2	1	0	1	0
18	imv	357.7	MF	1	INF	8	2	0	0	0
19	ipc_i1	1.5	UR	1	FIN	0	0	0	0	0
20	ipc_ip1	3383.5	UR	1	FIN	0	0	0	0	0
21	ipc_p1	1356.4	UR	1	FIN	0	0	0	0	0
22	ipc_s1	56.5	UR	1	FIN	0	0	0	0	0
23	ls37	245.3	UR	2	FIN	0	0	0	0	0
24	p38.nonstand	58.6	MF	1	N	0	5	0	0	0
25	p38.stand	1122.9	HM	1	N	0	4	0	1	0
26	sam	208.1	HM	1	FIN	0	3	0	0	0
27	set10	229.6	HM	1	P	0	0	0	1	0
28	set10	341.9	HM	2	INF	0	0	0	1	0
29	set11	109.5	HM	1	P	0	1	0	0	0
30	set11	297.9	HM	2	INF	0	1	0	0	0
31	set12	57.6	HM	1	P	0	2	0	0	0
32	set12	188.7	HM	2	INF	0	2	0	0	0
33	set13	277.1	HM	1	P	0	1	0	1	0
34	set13	186.4	HM	2	INF	0	0	0	0	0

(continued on next page)

Table 8.4: Model contradictions by each components (continued)

No.	Problem	Proof		Model		Model Backtrackings				
						F	N	T	M	U
		Time	Type	No	Type	L	P	R	P	R
35	set14	272.6	HM	1	P	0	0	0	1	0
36	set14	389.1	HM	2	INF	0	0	0	1	0
37	set15	228.4	HM	1	P	6	5	0	0	0
38	set15	513.4	HM	2	INF	2	4	0	0	0
39	set16	58.5	HM	1	P	0	2	0	0	0
40	set16	172.6	HM	2	INF	0	2	0	0	0
41	set17	175.8	HM	1	P	0	0	0	1	0
42	set17	577.6	HM	2	INF	0	0	0	1	0
43	set18	277.3	HM	1	P	0	0	0	1	0
44	set18	633.5	HM	2	INF	0	0	0	1	0
45	set19	335.9	HM	1	P	0	0	0	1	0
46	set19	704.8	HM	2	INF	0	1	0	1	0
47	set1	183.1	HM	1	P	0	2	0	0	0
48	set1	401.8	HM	2	INF	0	0	0	1	0
49	set20	226.8	HM	1	P	0	0	0	1	0
50	set21	297.1	HM	1	P	0	0	0	1	0
51	set21	419.9	HM	2	INF	0	0	0	1	0
52	set22	295.3	HM	1	P	0	0	0	1	0
53	set22	416.2	HM	2	INF	0	0	0	1	0
54	set23	0.6	SP	1	P	0	0	0	0	0
55	set23	0.6	SP	2	INF	0	0	0	0	0
56	set24	55.5	HM	1	P	0	1	0	0	0
57	set24	142.7	HM	2	INF	0	1	0	0	0
58	set25	0.6	SP	1	P	0	0	0	0	0
59	set25	0.6	SP	2	INF	0	0	0	0	0
60	set26	318.2	HM	1	P	0	0	0	1	0
61	set27	55.7	HM	1	P	0	1	0	0	0
62	set27	19.3	HM	2	INF	0	0	0	0	0
63	set28	56.0	HM	1	P	0	1	0	0	0
64	set28	31.4	HM	2	INF	0	0	0	0	0
65	set29	2.5	HM	1	P	0	0	0	0	0
66	set29	33.4	HM	2	INF	0	0	0	0	0
67	set2	54.9	HM	1	P	0	0	0	0	0
68	set2	178.1	HM	2	INF	0	0	0	0	0

(continued on next page)

Table 8.4: Model contradictions by each components (continued)

No.	Problem	Proof		Model		Model Backtrackings				
						F	N	T	M	U
		Time	Type	No	Type	L	P	R	P	R
69	set30	176.2	HM	1	P	0	0	0	1	0
70	set30	570.6	HM	2	INF	0	0	0	1	0
71	set35	56.1	HM	1	P	1	1	0	0	0
72	set35	182.8	HM	2	INF	0	1	0	0	0
73	set36	0.1	US	1	P	0	0	0	0	0
74	set36	0.1	US	2	INF	0	0	0	0	0
75	set3	57.2	HM	1	P	0	2	0	0	0
76	set3	184.7	HM	2	INF	0	2	0	0	0
77	set40	208.6	HM	1	P	0	1	0	1	0
78	set40	556.2	HM	2	INF	0	0	0	1	0
79	set41	54.3	HM	1	P	0	0	0	0	0
80	set41	111.0	HM	2	INF	0	0	0	0	0
81	set4	57.5	HM	1	P	0	2	0	0	0
82	set4	183.8	HM	2	INF	0	2	0	0	0
83	set5	235.5	HM	1	P	0	1	0	1	0
84	set5	630.6	HM	2	INF	0	0	0	1	0
85	set6	331.3	HM	1	P	0	1	0	1	0
86	set6	624.3	HM	2	INF	0	0	0	1	0
87	set7	324.9	HM	1	P	0	1	0	1	0
88	set7	738.0	HM	2	INF	0	0	0	1	0
89	set8	234.0	HM	1	P	0	1	0	1	0
90	set8	548.8	HM	2	INF	0	0	0	1	0
91	set9	230.8	HM	1	P	0	0	0	1	0
92	set9	637.8	HM	2	INF	0	0	0	1	0
93	t10	330.1	HM	1	P	0	1	0	1	0
94	t11	792.5	HM	1	P	14	6	0	0	1
95	t11	14540.7	—	1	INF	0	0	0	0	0
96	t13	158.8	HM	1	P	0	7	0	0	0
97	t13	27499.5	HM	1	INF	0	0	0	1	0
98	t1	1867.8	UR	1	P	103	21	0	0	3
99	t1	2337.9	UR	1	INF	0	0	0	0	0
100	t2	77.1	HM	1	P	0	0	0	0	0
101	t2	912.3	HM	1	INF	0	0	0	0	0
102	t3	78.0	HM	1	P	0	0	0	0	0

(continued on next page)

Table 8.4: Model contradictions by each components (continued)

No.	Problem	Proof		Model		Model Backtrackings				
						F	N	T	M	U
		Time	Type	No	Type	L	P	R	P	R
103	t3	914.6	HM	1	INF	0	0	0	0	0
104	t6	11018.4	—	1	P	10	3	0	0	0
105	t6	2015.7	HM	1	INF	0	1	0	0	0
106	t7	4659.8	HM	1	INF	0	1	0	0	0
107	tl10	240.9	HM	1	P	6	0	0	1	0
108	tl10	358.4	HM	1	INF	0	0	0	1	0
109	tl11	240.4	HM	1	P	6	0	0	1	0
110	tl11	358.1	HM	1	INF	0	0	0	1	0
111	tl12	253.8	HM	1	P	6	0	0	1	0
112	tl12	310.2	MF	1	INF	4	3	0	0	0
113	tl13	255.4	HM	1	P	6	0	0	1	0
114	tl13	448.2	MF	1	INF	1	7	0	0	0
115	tl14	253.6	HM	1	P	6	0	0	1	0
116	tl14	444.4	HM	1	INF	1	5	0	1	0
117	tl15	518.9	MF	1	P	3	4	0	0	0
118	tl15	378.2	MF	1	INF	3	5	0	0	0
119	tl16	300.8	HM	1	P	5	0	0	1	0
120	tl16	381.4	HM	1	INF	0	0	0	1	0
121	tl17	305.6	HM	1	P	5	0	0	1	0
122	tl17	329.5	HM	1	INF	0	0	0	1	0
123	tl18	1834.1	HM	1	INF	0	0	0	1	0
124	tl19	219.4	HM	1	P	2	0	0	1	0
125	tl19	246.8	HM	1	INF	0	0	0	1	0
126	tl1	149.6	HM	1	P	2	1	0	0	0
127	tl1	265.5	HM	1	INF	0	1	0	0	0
128	tl23	194.7	HM	1	P	2	0	0	1	0
129	tl23	200.9	MF	1	INF	0	4	0	0	0
130	tl24	489.6	HM	1	P	6	0	0	1	0
131	tl24	397.4	HM	1	INF	0	0	0	1	0
132	tl25	2836.9	—	1	INF	0	0	0	0	0
133	tl26	582.5	HM	1	P	6	0	0	1	0
134	tl26	305.9	HM	1	INF	0	0	0	1	0
135	tl27	862.3	HM	1	P	12	0	0	1	0
136	tl27	522.7	HM	1	INF	0	0	0	1	0

(continued on next page)

Table 8.4: Model contradictions by each components (continued)

No.	Problem	Proof		Model		Model Backtrackings				
						F	N	T	M	U
		Time	Type	No	Type	L	P	R	P	R
137	tl2	171.3	HM	1	P	2	0	0	1	0
138	tl2	269.8	HM	1	INF	0	0	0	1	0
139	tl30	637.4	HM	1	P	12	0	0	1	0
140	tl30	646.8	HM	1	INF	2	3	0	1	0
141	tl31	269.9	HM	1	P	6	0	0	1	0
142	tl31	394.9	HM	1	INF	0	0	0	1	0
143	tl32	408.9	HM	1	P	12	0	0	1	0
144	tl32	525.2	HM	1	INF	0	0	0	1	0
145	tl33	404.7	HM	1	P	12	0	0	1	0
146	tl33	524.0	HM	1	INF	0	0	0	1	0
147	tl34	885.0	HM	1	P	12	0	0	1	0
148	tl34	544.6	HM	1	INF	0	0	0	1	0
149	tl35	405.3	HM	1	P	12	0	0	1	0
150	tl35	522.0	HM	1	INF	0	0	0	1	0
151	tl37	972.6	HM	1	P	12	0	0	1	0
152	tl37	628.8	HM	1	INF	0	0	0	1	0
153	tl38	456.1	HM	1	P	12	0	0	1	0
154	tl38	591.1	HM	1	INF	0	0	0	1	0
155	tl3	255.2	HM	1	P	2	0	0	1	0
156	tl3	281.6	HM	1	INF	0	0	0	1	0
157	tl4	190.6	HM	1	P	2	0	0	1	0
158	tl4	221.2	HM	1	INF	0	0	0	1	0
159	tl5	181.8	HM	1	P	2	0	0	1	0
160	tl5	197.0	MF	1	INF	2	2	0	0	0
161	tl6	239.1	HM	1	P	6	0	0	1	0
162	tl6	356.3	HM	1	INF	0	0	0	1	0
163	tl7	485.8	HM	1	P	6	0	0	1	0
164	tl7	297.1	HM	1	INF	0	1	0	1	0
165	tl8	272.4	HM	1	P	6	0	0	1	0
166	tl8	235.5	HM	1	INF	0	0	0	1	0
167	tl9	237.4	HM	1	P	6	0	0	1	0
168	tl9	356.6	HM	1	INF	0	0	0	1	0
169	wos15	250.4	UR	1	INF	0	0	0	0	0
170	wos19	285.2	HM	2	FIN	0	0	1	0	0

(continued on next page)

Table 8.4: Model contradictions by each components (continued)

No.	Problem	Proof		Model		Model Backtrackings				
						F	N	T	M	U
		Time	Type	No	Type	L	P	R	P	R
171	wos21	325.1	UR	1	FIN	0	0	0	0	0
172	wos31_EXQ1	97.5	HM	1	INF	0	2	0	0	0
173	wos31_EXQ2	119.9	MF	1	INF	12	9	0	0	0
174	wos31_EXQ3	90.9	MF	1	INF	9	17	0	0	0

Chapter 9

Conclusion

9.1 Summary

In this dissertation we have presented semantic hyper-linking to use semantics in an instance-based theorem prover. Semantic hyper-linking can be seen as a combination of Gilmore's procedure and hyper-linking, with the assistance of semantics to prune the search space. The process is basically a model searching procedure to look for a model for the input clauses. New ground clauses are generated by hyper-linking and enumeration of the Herbrand base. Several model finding strategies are also presented to advance the proof searching without expensive semantic checks. Two other major methods, rough resolution and UR resolution, are also used. The basic semantic hyper-linking method is good at finding small non-Horn part of the proof; rough resolution is used to avoid generating large literals needed for the proof (the large part of the proof); UR resolution is also used to solve the Horn part of the proof.

One major advantage of our methods is that completeness is independent of the input semantics and this gives much flexibility in practical use.

A prover `CLIN-S` has been implemented and has obtained fast proofs, with no other human guidance, on some hard problems that are difficult or impossible for other provers. Some fast proofs with trivial semantics also show that semantic hyper-linking is a powerful syntactic proving technique as well. We believe that a natural semantics is often more effective in pruning the search space—just like diagrams are helpful for mathematicians to prove theorems. However, in practice expensive semantic checks often offset such advantages.

9.2 Extensions

9.2.1 Built-in Specialized Decision Procedures

Often built-in specialized decision procedures are needed for decidable semantics. We have discussed the use of SUP-INF procedure for semantics expressed in Bledsoe real arithmetics. In general, more specialized decision procedures can be built in to allow the use of decidable semantics in other domains such as ordinary algebra and geometry [Tar51], a subclass of elementary geometry [Wt78].

Again if the natural semantics is not available, either due to lack of a decision procedure or no such procedure is built-in, one can always choose to use trivial or approximate semantics (see Chapter 6).

9.2.2 Model Finding

This is one aspect of our prover that we think more improvement can be done. The idea of model finding is simple: we look for a better model by detecting some contradictions, direct or indirect, from a ground model M as early as possible. To find contradicted instances is not trivial but often rewarding since a better model often means the proof search being "closer" to complete a semantic tree. If there are more model finding strategies, even better models could be found and save the effort for semantic hyper-linking, particularly the expensive semantic checks can be avoided. Possibly special decision procedures can be used here also.

9.2.3 Refining Rough Resolution

Rough resolution is powerful enough to help semantic hyper-linking prove some hard theorems which cannot be obtained otherwise. However we have observed that likely maximal literals are the source of rapid search space expansion in some hard theorems like LIM4 and LIM5. Future research includes further investigation of the role of likely maximal literals and avoid generating unnecessary resolvents from resolving upon likely maximal literals. One possible direction is applying rough resolution idea on paramodulation since equality axioms often contain likely maximal literals. Also, focusing on relevant resolvents should be another important issue to be addressed.

9.2.4 Many-Sorted Semantics

In some situations it is better to express objects of different *types* (or *sorts*) and specify functions and predicates similarly. Such *many-sorted semantics* has the obvious advantage to strictly restrict the formation of terms and literals, especially in the enumeration of the Herbrand base. For example, for a modal logic (possible-worlds logic) theorem transformed to first-order logic, we might introduce a type called *world* for all the worlds. Thus the accessibility relation [Kri63, Kri71] $r(X, Y)$ only takes arguments X and Y of type *world*. Preliminary results have shown that using many-sorted semantics can help `CLIN-S` prove some problems faster.

Appendix A

Problem Set

In this appendix, we list the problem set we have run on `CLIN`-S, an implementation of semantic hyper-linking and related techniques. We do not list the input clauses here. Instead, for each problem, we list the names, brief descriptions, and the related references where the original sources or detailed descriptions of the problem can be found. An excellent source of a list of problems for automated first-order theorem provers is the TPTP (Thousands of Problems for Theorem Provers) problem library [SSY93].

Name	*Description*	*References*
AM8	The attaining maximum (or minimum) value theorem in real analysis: a continuous function f is a closed real interval $[a, b]$ attains its minimum (or maximum) in this interval.	[Ble83]
AND	Andrew's challenge problem: $[(\exists x)(\forall y)(p(x) \equiv p(y)) \equiv ((\exists x)q(x) \equiv (\forall y)q(y))] \equiv [(\exists x)(\forall y)(q(x) \equiv q(y)) \equiv ((\exists x)p(x) \equiv (\forall y)p(y))]$.	[Pel86]

EXQ1	Derive a contradiction from the conjunction of: (1) $m \neq n$, (2) $n \neq k$, (3) $k \neq m$, (4) $y = m \vee [f(y,m) \equiv (\exists z)(z \neq m \wedge z \neq y \wedge f(y,z) \wedge f(z,y))]$, (5) $y = n \vee [f(y,n) \equiv \neg(\exists z)(z \neq n \wedge z \neq y \wedge f(y,z) \wedge f(z,y))]$, (6) $y = k \vee [f(y,k) \equiv (y = m \vee y = n)]$. Of the 23 clauses obtained from the conjunction, 11 are non-Horn clauses.	[Wan65]
HALT	The undecidability of the halting problem encoded in first-order logic.	[Bru91]
IMV	It states: If function f is continuous in a real closed interval $[a,b]$, where $f(a) \leq 0$ and $0 \leq f(b)$, then $\exists x f(x) = 0$. A second order (in which functions might be quantified) axiom, *least upper bound* (LUB) axiom, is needed. LUB states: Every non-empty, bounded from above set A has a least upper bound. We use a first-order instantiation is obtained by instantiating set variable A by the set $\{ z : z \leq b \wedge f(z) \leq 0 \}$. More detailed discussions can be found in the references.	[Ble83, BB82, WB87]
EXQ2	Replace (2) and (3) in EXQ1 by (2') $(n = k \vee k = m)$ and (3') $y = j \vee (f(y,j) \equiv y = k)$.	[Wan65]
EXQ3	Drop (2') from EXQ2.	[Wan65]
LIM1–3 (Lim+)	The theorems (proposed by Bledsoe) in calculus that the sum of two continuous functions is continuous. There are harder LIM4–5, LIM5= and LIM5=h (expressed in equalities).	[Ble90]
SYN036-4	Andrew's challenge problem in different clauses.	[Pel86]
SYN038-1	Variant of ex4t1.	[SWKL74]
SYN086-1	Variant of Andrew's challenge problem.	[Pel86]

SYN086-2	Variant of Andrew's challenge problem.	[Pel86]
am8	See AM8.	
andrew	See AND.	
bledsoe1–3 (Lim+)	They are LIM1–3 respectively. See LIM1–3.	
ex4t1, ex4t2	$\exists x \exists y \forall z(((p(x,y) \supset (p(x,z) \equiv q(y,z))) \wedge (p(x,y) \equiv (p(z,z) \supset q(z,z)))) \supset (q(z,y) \equiv q(z,z))))$	[SWKL74]
ex6t1, ex6t2	For all x in a ring, $x * e = e$ where e is the additive identity.	[SWKL74]
halting	See HALT.	
imv	See IMV.	
ipc	The single axioms in the Tarski-Bernays systems for reasoning in the implicational propositional calculus by Lukasiewicz. Such systems consists of three axioms, simplification (S), Peirce's Law (P), hypothetical syllogism (H) as well as inference rules substitution and detachment (modus ponens). There are single axioms, L_1, L_2, L_3, which allow one to derive all of the Tarski-Bernays axioms and hence complete.	[Luk48, Pfe88]
ipc_i1	To prove that in the implicational propositional calculus (see ipc) the axiom $I : p \supset p$ logically follows from the single axiom $L_1 : ((p \supset q) \supset r) \supset ((r \supset p) \supset (s \supset p))$.	[Luk48, Pfe88]
ipc_ip1	To prove that in the implicational propositional calculus (see ipc) the Peirce's Law axiom $P : ((p \supset q) \supset p) \supset p$ logically follows from the single axiom $L_1 : ((p \supset q) \supset r) \supset ((r \supset p) \supset (s \supset p))$ and axiom $I : p \supset p$.	[Luk48, Pfe88]

ipc_p1	To prove that in the implicational propositional calculus (see ipc) the Peirce's Law axiom $P : ((p \supset q) \supset p) \supset p$ logically follows from the single axiom $L_1 : ((p \supset q) \supset r) \supset ((r \supset p) \supset (s \supset p))$.	[Luk48, Pfe88]
ipc_s1	To prove that in the implicational propositional calculus (see ipc) the simplification axiom $S : p \supset (q \supset p)$ logically follows from the single axiom $L_1 : ((p \supset q) \supset r) \supset ((r \supset p) \supset (s \supset p))$.	[Luk48, Pfe88]
ls37	For all x in a ring R, $x * 0 = 0$ where 0 is the additive identity.	[MOW76]
p38	The 38th problem of Pelletier's 75 problems for automated theorem provers: $(\forall x)[p(a) \wedge (p(x) \supset (\exists y)(p(y) \wedge r(x,y))) \supset (\exists z)(\exists w)(p(z) \wedge r(x,w) \wedge r(w,z))] \equiv$ $(\forall x)[(\neg p(a) \vee p(x) \vee (\exists z)(\exists w)(p(z) \wedge r(x,w) \wedge r(w,z))) \wedge (\neg p(a) \vee \neg(\exists y)(p(y) \wedge r(x,y)) \vee (\exists z)(\exists w)(p(z) \wedge r(x,w) \wedge r(w,z)))]$	[Pel86]
sam	A lemma first proved in the semi-automated mathematics: Let L be a modular lattice with 0 and 1. Suppose that A and B are elements of L such that $(A \vee B)$ and $(A \wedge B)$ both have not necessarily unique complements. Then, $(A \vee B)' = ((A \vee B)' \vee ((A \wedge B)' \wedge B)) \wedge ((A \vee B)' \vee ((A \wedge B)' \wedge A))$.	[GOBS69, MOW76]
set1	The set theorem: $\forall X, Y(X \cup Y = Y \cup X)$. We use a reduced axiom set.	[BLM+86]
set2	The set theorem: $\forall X, Y(X \cap Y = Y \cap X)$. We use a reduced axiom set.	[BLM+86]
set3	The set theorem: $\forall X(X \cup X = X)$. We use a reduced axiom set.	[BLM+86]
set4	The set theorem: $\forall X(X \cap X = X)$. We use a reduced axiom set.	[BLM+86]
set5	The set theorem: $\forall X(X \cup \neg X = universe)$. We use a reduced axiom set.	[BLM+86]

set6	The set theorem: $\forall X(X \cap \neg X = \emptyset)$. We use a reduced axiom set.	[BLM+86]
set7	The set theorem: $\neg universe = \emptyset$. We use a reduced axiom set.	[BLM+86]
set8	The set theorem: $\neg\emptyset = universe$. We use a reduced axiom set.	[BLM+86]
set9	The set theorem: $\forall X(\neg\neg X = X)$. We use a reduced axiom set.	[BLM+86]
set10	The set theorem: $\forall X(X \cup \emptyset = X)$. We use a reduced axiom set.	[BLM+86]
set11	The set theorem: $\forall X(X \cap universe = X)$. We use a reduced axiom set.	[BLM+86]
set12	The set theorem: $\forall X(X \cup universe = universe)$. We use a reduced axiom set.	[BLM+86]
set13	The set theorem: $\forall X(X \cap \emptyset = \emptyset)$. We use a reduced axiom set.	[BLM+86]
set14	The set theorem: $\forall X, Y, Z((X \cup Y) \cup Z = X \cup (Y \cup Z))$. We use a reduced axiom set.	[BLM+86]
set15	The set theorem: $\forall X, Y, Z((X \cap Y) \cap Z = X \cap (Y \cap Z))$. We use a reduced axiom set.	[BLM+86]
set16	The set theorem: $\forall X, Y(X \subseteq Y \supset X \cap Y = X)$. We use a reduced axiom set.	[BLM+86]
set17	The set theorem: $\forall X, Y(\neg(X \cup Y) = \neg X \cap \neg Y)$. We use a reduced axiom set.	[BLM+86]
set18	The set theorem: $\forall X, Y(\neg(X \cap Y) = \neg X \cup \neg Y)$. We use a reduced axiom set.	[BLM+86]
set19	The set theorem: $\forall X, Y((X \cup Y) \cap (X \cup \neg Y) = X)$. We use a reduced axiom set.	[BLM+86]
set20	The set theorem: $\forall X, Y(X - Y = X \cap \neg Y)$. We use a reduced axiom set.	[BLM+86]
set21	The set theorem: $\forall X, Y, Z(X \cap (Y \cup Z) = X \cap Y \cup X \cap Z)$. We use a reduced axiom set.	[BLM+86]
set22	The set theorem: $\forall X, Y, Z(X \cup Y \cap Z = (X \cup Y) \cap (X \cup Z))$. We use a reduced axiom set.	[BLM+86]

set23	The set theorem: $\forall X(\emptyset \subseteq X)$. We use a reduced axiom set.	[BLM+86]
set24	The set theorem: $\forall X, Y, Z(X \subseteq Y \wedge Y \subseteq Z \supset X \subseteq Z)$. We use a reduced axiom set.	[BLM+86]
set25	The set theorem: $\forall X, Y(X \subseteq Y \supset X \in 2^Y)$. We use a reduced axiom set.	[BLM+86]
set26	The set theorem: $\forall X, Y(X \bar{\cap} Y \supset X \cap Y = \emptyset)$. We use a reduced axiom set.	[BLM+86]
set27	The set theorem: $\forall X, Y(X \subseteq (X \cup Y))$. We use a reduced axiom set.	[BLM+86]
set28	The set theorem: $\forall X, Y((X - Y) \subseteq X)$. We use a reduced axiom set.	[BLM+86]
set29	The set theorem: $\forall X, Y, Z(X \subseteq (Y \cap Z) \supset X \subseteq Y \wedge X \subseteq Z)$. We use a reduced axiom set.	[BLM+86]
set30	The set theorem: $\forall X, Y(2^{X \cap Y} = 2^X \cap 2^Y)$. We use a reduced axiom set.	[BLM+86]
set31	The set theorem: $\forall X, Y, Z(X \times (Y \cap Z) \subseteq ((X \times Y) \cap (X \times Z)))$. We use a reduced axiom set.	[BLM+86]
set32	The set theorem: $\forall X, Y, Z, W((X \subseteq Y) \cap (Z \subseteq W) \supset X \times Z \subseteq Y \times W)$. We use a reduced axiom set.	[BLM+86]
set33	The set theorem: $\forall X, Y, Z, W(meq(a, b) \wedge meq(c, d) \supset < a, c >=< b, d >)$. We use a reduced axiom set.	[BLM+86]
set34	The set theorem: $\forall X, Y, Z(X =< Y, Z >\supset opp(X))$. We use a reduced axiom set.	[BLM+86]
set35	The set theorem: $\forall X, Y(m(X) \wedge m(Y) \supset \{X\} \subseteq \{X, Y\})$. We use a reduced axiom set.	[BLM+86]
set36	The set theorem: $\forall X, Y(m(X) \wedge m(Y) \supset \{X, Y\} = \{Y, X\})$. We use a reduced axiom set.	[BLM+86]
set40	The set theorem: $universe = 2^{universe}$. We use a reduced axiom set.	[BLM+86]
set41	The set theorem: $(\forall X, Y(X \subseteq Y \supset X \in 2^Y)) \supset (\forall X(m(X)))$. We use a reduced axiom set.	[BLM+86]

t1	Betweenness is symmetric.	[MOW76]
t2	$\forall X, Y$ (X is between X and Y)	[MOW76]
t3	$\forall X, Y$ (Y is between X and Y)	[MOW76]
t6	For any three distinct points X, Y, and Z, if Y is between X and Z, then both X is not between Y and Z and Z is not between X and Y.	[MOW76]
t7	For all pairs of distinct points Y and Z, if W and X are on the line YZ to the left of Y, then either W is between X and Y or X is between W and Y.	[MOW76]
t10	If X, Y, and Z are collinear in one order, they are collinear in any order.	[MOW76]
t11	Given the Lower Dimension Axiom with three points $c1$, $c2$ and $c3$: $c2$ is not between $c1$ and $c3$, $c3$ is not between $c2$ and $c1$, and $c1$ is not between $c3$ and $c2$. Then The three points $c1$, $c2$ and $c3$ specified are not collinear.	[MOW76]
t13	If $Z1$, $Z2$, and $Z3$ are each collinear with distinct points X and Y, then $Z1$, $Z2$, and $Z3$ are collinear.	[MOW76]
t1–38	Propositional temporal logic (PTL) problems taken from [Pla86, Ost89]. The alphabet of PTL consists of: propositional symbols; connectives: **T**, **F**, $\neg$, $\vee$, $\wedge$, $\supset$, and $\equiv$; modal operators: $\Box$ ("henceforth"), $\Diamond$ ("eventually"), *bigcirc* ("in the next state"), and **U** ("until"); and auxiliary symbols: "(" and ")".	[Pla86, Ost89]
t1	$w \rightarrow \Diamond w$	[Pla86, Ost89]
t2	$\bigcirc w \rightarrow \Diamond w$	[Pla86, Ost89]
t3	$\Box w \equiv \Box\Box w$	[Pla86, Ost89]
t4	$\Diamond w \equiv \Diamond\Diamond w$	[Pla86, Ost89]
t5	$\Diamond\neg w \equiv \neg\Box w$	[Pla86, Ost89]
t6	$\Box(w_1 \rightarrow w_2) \rightarrow (\Diamond w_1 \rightarrow \Diamond w_2)$	[Pla86, Ost89]

t7	$\Box(w_1 \wedge w_2) \equiv (\Box w_1) \wedge (\Box w_2)$	[Pla86, Ost89]
t8	$\Diamond(w_1 \vee w_2) \equiv (\Diamond w_1) \vee (\Diamond w_2)$	[Pla86, Ost89]
t9	$(\Box w_1 \vee \Box w_2) \rightarrow \Box(w_1 \vee w_2)$	[Pla86, Ost89]
t10	$\Diamond(w_1 \wedge w_2) \rightarrow (\Diamond w_1 \wedge \Diamond w_2)$	[Pla86, Ost89]
t11	$(\Box w_1 \wedge \Diamond w_2) \rightarrow \Diamond(w_1 \wedge w_2)$	[Pla86, Ost89]
t12	$\bigcirc(w_1 \wedge w_2) \equiv (\bigcirc w_1 \wedge \bigcirc w_2)$	[Pla86, Ost89]
tl13	$\bigcirc(w_1 \vee w_2) \equiv (\bigcirc w_1 \vee \bigcirc w_2)$	[Pla86, Ost89]
tl14	$\bigcirc(w_1 \rightarrow w_2) \equiv (\bigcirc w_1 \rightarrow \bigcirc w_2)$	[Pla86, Ost89]
tl15	$\bigcirc(w_1 \equiv w_2) \equiv (\bigcirc w_1 \equiv \bigcirc w_2)$	[Pla86, Ost89]
tl16	$\bigcirc\Box w \equiv \Box \bigcirc w$	[Pla86, Ost89]
tl17	$\bigcirc\Diamond w \equiv \Diamond \bigcirc w$	[Pla86, Ost89]
tl18	$\Box\Diamond\Box w \equiv \Diamond\Box w$	[Pla86, Ost89]
tl19	$\Diamond\Box\Diamond w \equiv \Box\Diamond w$	[Pla86, Ost89]
tl20	$\Box w \equiv (w \wedge \bigcirc\Box w)$	[Pla86, Ost89]
tl21	$\Diamond w \equiv (w \vee \bigcirc\Diamond w)$	[Pla86, Ost89]
tl22	$(w \wedge \Diamond\neg w) \rightarrow \Diamond(w \wedge \bigcirc\neg w)$	[Pla86, Ost89]
tl23	$(\neg w)\mathbf{U}w \equiv \Diamond w$	[Pla86, Ost89]
tl24	$\Box w_1 \wedge \Diamond w_2 \rightarrow w_1\mathbf{U}w_2$	[Pla86, Ost89]
tl25	$(w_1\mathbf{U}w_2)\mathbf{U}w_2 \rightarrow (w_1\mathbf{U}w_2)$	[Pla86, Ost89]
tl26	$w_1\mathbf{U}(w_1\mathbf{U}w_2) \rightarrow (w_1\mathbf{U}w_2)$	[Pla86, Ost89]
tl27	$(\Box w_1 \wedge (w_2\mathbf{U}w_3)) \rightarrow (w_1 \wedge w_2)\mathbf{U}(w_1 \wedge w_3)$	[Pla86, Ost89]
tl28	$(\bigcirc w_1\mathbf{U} \bigcirc w_2) \equiv \bigcirc(w_1\mathbf{U}w_2)$	[Pla86, Ost89]
tl29	$(w_1 \wedge w_2)\mathbf{U}w_3 \equiv (w_1\mathbf{U}w_3) \wedge (w_2\mathbf{U}w_3)$	[Pla86, Ost89]
tl30	$w_1\mathbf{U}(w_2 \vee w_3) \equiv (w_1\mathbf{U}w_2) \vee (w_1\mathbf{U}w_3)$	[Pla86, Ost89]
tl31	$(\Diamond w_1 \vee \Diamond w_2) \supset (((\neg w_1)\mathbf{U}w_2) \vee ((\neg w_2)\mathbf{U}w_1))$	[Pla86, Ost89]

tl32	$w_1\mathbf{U}(w_2 \wedge w_3) \supset (w_1\mathbf{U}w_2) \wedge (w_1\mathbf{U}w_3)$	[Pla86, Ost89]
tl33	$(w_1\mathbf{U}w_3) \vee (w_2\mathbf{U}w_3) \supset (w_1 \vee w_2)\mathbf{U}w_3$	[Pla86, Ost89]
tl34	$(w_1 \supset w_2)\mathbf{U}w_3 \supset (w_1\mathbf{U}w_3 \supset w_2\mathbf{U}w_3)$	[Pla86, Ost89]
tl35	$(w_1\mathbf{U}w_2) \wedge ((\neg w_2)\mathbf{U}w_3) \supset w_1\mathbf{U}w_3$	[Pla86, Ost89]
tl36	$(w_1\mathbf{U}(w_2 \wedge w_3)) \supset ((w_1 \supset w_2)\mathbf{U}w_3)$	[Pla86, Ost89]
tl37	$(w_1\mathbf{U}w_2)\mathbf{U}w_3 \supset (w_1 \vee w_2)\mathbf{U}w_3$	[Pla86, Ost89]
tl38	$w_1\mathbf{U}(w_2\mathbf{U}w_3) \supset (w_1 \vee w_2)\mathbf{U}w_3$	[Pla86, Ost89]
wos15	The closure property of subgroups: If O is a subgroup of G, then for all x and y in O, $x * y$ is still in O.	[MOW76]
wos19	Subgroups of index 2 are normal: If O is a subgroup of G and there are exactly 2 cosets in G/O, then O is normal (for all x in G and y in O, $x * y * x^{-1}$ is in O).	[MOW76]
wos20	A variant of wos19.	[MOW76]
wos21	A variant of ls37.	[MOW76]
wos31_EXQ1-3	See EXQ1-3.	

Appendix B

CLIN-S User's Manual

B.1 Introduction

This manual documents the use of `CLIN-S`, a *semantic hyper-linking* theorem prover developed by Heng Chu and David A. Plaisted at the Computer Science Department of University of North Carolina at Chapel Hill. `CLIN-S` is a refutational clause-form theorem prover and has been developed based on `CLIN` (the *hyper-linking* theorem prover) by Shie-Jue Lee, Geoffrey D. Alexander, and David A. Plaisted [Lee90, LP92]. Although `CLIN-S` and `CLIN` share the same input and output formats (or even part of the name), the underlying philosophy and searching strategies between hyper-linking and semantic hyper-linking are different. In fact, `CLIN-S` can conduct either hyper-linking (the default) or semantic hyper-linking proof searching. Thus `CLIN-S` is a superset of `CLIN` and semantic hyper-linking is invoked when an example (a semantics) is read in as part of the input. However, *in this manual we only focus on how to use* `CLIN-S` *to prove theorems with semantic hyper-linking.*

We assume the users have a knowledge of refutational clause-based first-order theorem proving and are familiar with the commonly used terminologies including *atom, function, variable, term, predicate, literal, clause, and model.* For an introduction to first-order theorem proving, please refer to [CL73, Lov78]. For more detailed discussion on hyper-linking or semantic hyper-linking, please refer to [CP93a, CP93c, CP94a, Lee90, LP92].

`CLIN-S` is an instance-based theorem prover. The axioms and the negated theorem are converted to *clauses*[1] before submitted to `CLIN-S` to search for the proof. Proofs are found by refutation, namely, axioms and *negated* theorems are given as the input for `CLIN-S` to derive a contradiction. Ground instances of input clauses are generated and a proof is found if an unsatisfiable ground instance set is generated. If no semantic information (examples) is specified, the *hyper-linking* technique [Lee90, LP92] is used and the search for the proof is achieved by using the hyper-linking operation to generate fully linked unsatisfiable ground instance set. If an example (the semantic information) is given, *semantic hyper-linking* is used and the search for the proof is done by *model finding*, namely exhaustively searching for models of the input clause set during which systematic searching for ground models of the ground instance set provides guidance to the prover.

`CLIN-S` is a superset of `CLIN` and is compatible with `CLIN`: all input formats, flags and commands for `CLIN` are also available in `CLIN-S`. `CLIN-S` has new flags and commands to control some of semantic hyper-linking features. Users are referred to the `CLIN` user's manual [LC92] for flags and commands to control hyper-linking.

Unlike `CLIN`, `CLIN-S` has been developed and is now only available for Quintus Prolog. Versions for other Prolog systems might be available in the future. However, C-Prolog will not be supported because it lacks linking ability with C programs which is an important part of the finite model checker of `CLIN-S` (see Section B.2.3).

To use the semantic hyper-linking technique in `CLIN-S`, one has to provide semantic information (or an example) for the input clauses. A *natural* or *intended* semantics is a natural example for the input axioms and theorem. If the natural semantics is not readily available, one can always use a *trivial* semantics. For example, a natural semantics for theorems in group theory could be a real example of groups such as $\mathbf{Z}_4$ (set of integers modulo 4); a trivial semantics is one in which all true literals are exactly the positive ones. Details of specifying semantic information will be discussed in Section B.6.

There are other syntactic techniques used with semantic hyper-linking to provide fast inferences without expensive semantic checks. Those techniques include *model finding strategies (filtering, natural and minimal replacement, simple term rewriting and UR resolution)* [CP93b] and *rough resolution* [CP93c].

[1] Input is in *clause* or *sprfn* format [LC92].

B.2 Overview of `CLIN-S`

`CLIN-S` can work exactly like `CLIN` using the hyper-linking proving technique to conduct syntax based search. `CLIN-S` can also apply semantic hyper-linking and use the semantic information furnished by the users to guide the search for the proofs. The basic procedure of either technique is generating ground instances of the input clauses until an unsatisfiable ground set is obtained. It is important to know that semantic hyper-linking is complete and is *independent* of the input semantics. So a proof for an unsatisfiable clause set can always be found with any given semantics, and users have the freedom to use different semantics although better semantics usually gives better results.

In this section, we will discuss how `CLIN-S` works. We will introduce briefly hyper-linking and semantic hyper-linking. We will also discuss some non-semantic procedures that collaborate with semantic hyper-linking to find proofs even faster.

B.2.1 Hyper-Linking

Hyper-linking is the major operation in `CLIN`. With a chosen clause (the *nucleus*), hyper-linking *links*—through unification—*complementary literals* from the nucleus and other clauses. This is in spirit similar to the connection graph method. However, hyper-linking does not store the links which may require large space. Instead, after a proper substitution is applied, hyper-linking generates instances of the nucleus with *each* literal hyper-linked with one literal in some other clause.

Hyper-linking is a simple syntactic method. The proof search could be further controlled by using various support strategies, sliding priority, or predicate replacement. Their use is discussed in `CLIN` user's manual [LC92] and more details can also be found in [Lee90].

B.2.2 Semantic Hyper-Linking

Semantic hyper-linking is an instance-based theorem prover that, with the guidance of an input semantics, searches for an unsatisfiable ground instance set of the input clauses. The input semantics is usually represented by an example which consists of the domain, ground and, if possible, non-ground interpretations for constants, functions, and predicates. During the proof searching, semantic hyper-linking generates "new" semantics implicitly from the input semantics and a model for the ground instance set. The proof searching terminates when the ground instance set is unsatisfiable and a proof is found.

Semantic hyper-linking is basically a model searching method—it searches for models for the input clauses. Once one such "model" is constructed, `CLIN-S` *contradicts* this model by generating ground instances false in this model. If the input clause set is unsatisfiable, such model searching process will terminate with an unsatisfiable ground instance set for which no model exists.

Two major processes in semantic hyper-linking are *ground instance generation* and *model finding*. Such two processes constitute a *round*. Each round begins with a semantics I as a "model" for the input clauses (initially it is the input semantics I_0). Then the ground instance generation process generates the smallest[2] ground instances that are false in this "model." Those new ground instances show that I is *not* a model for the input clauses and another new semantics I' should be constructed. The model finding process searches for a model M for the new ground instance set. I' is constructed implicitly from the input semantics I_0 and the ground model M. Another new round starts with I' which is now assumed to be a "model" for the input clause set. Such a loop of rounds continues until M does not exist for the ground instance set and a proof is found (based on Herbrand's theorem). The basic semantic hyper-linking procedure is described in Fig. B.1.

In the context of clause-form theorem proving, a semantics can be seen as a set of all literals true in this semantics or, equivalently, an interpretation of the atoms in the Herbrand base. Semantic hyper-linking intends to show that no such set (or interpretation) exists for an unsatisfiable clause set.

Each new semantics I is expressed *implicitly* using the input semantics I_0 (specified by an example) and the model M for the ground instance set obtained so far: M is the finite part of I and $I_0 - M$ is the infinite part. More precisely, if we only look at the largest subset $\mathcal{L}$ of M that contains literals false in I_0 (having different interpretation), I is the same as I_0 except for the literals in $\mathcal{L}$ (or the interpretation

[2]The ordering is based on the size of the largest literal in a clause.

of the corresponding atoms). We represent I as $I_0[\mathcal{L}]$. Let $\overline{\mathcal{L}}$ be the set $\{\neg L | L \in \mathcal{L}\}$, $I = I_0[\mathcal{L}]$ then is $\mathcal{L} \cup (I_0 - \overline{\mathcal{L}})$.

In the next two sections we will briefly describe those two main processes of `CLIN-S`.

B.2.2.1 Ground Instance Generation

Semantics is mainly used in ground instance generation. Given a semantics $I = I_0[\mathcal{L}]$, `CLIN-S` generates ground instances false in I to show that I is not a model for the input clauses. Any such instance C contains literals that are either

Type 1 literals: in $\overline{\mathcal{L}}$ (false in the finite part $\mathcal{L}$) or

Type 2 literals: $\overline{(I_0 - \mathcal{L})}$ (false in the infinite part $I_0 - \overline{\mathcal{L}}$).

Type 1 literals can be easily determined with a given $\mathcal{L}$. Type 2 literals can be determined using systematic enumeration of the Herbrand base.

Given a semantics, `CLIN-S` applies ground evaluation on a ground literal L to determine the truth value; non-ground evaluation is applied on a non-ground clause. Ground evaluation can be achieved relatively easily using computing procedures written in, for example, Prolog. Non-ground evaluation is always decidable if the domain is finite (which is equivalent to multiple ground evaluations with all variables properly instantiated with domain values). However, non-ground evaluation in general needs decision procedures if the semantics has an infinite domain. *Semantic checks* are referred to either ground or non-ground evaluation. A semantics is call *decidable* if non-ground evaluation is possible. `CLIN-S` has a built-in procedure for semantics in Bledsoe real arithmetic (see Section B.6.3.1).

The use of semantics in semantic hyper-linking mainly occurs in two places: One is after type 1 literals are determined and non-ground evaluation is used to prune the search space. In the other place ground evaluation is used in generating type 2 literals (or evaluation of any ground literals). In general, only ground evaluation is needed for `CLIN-S` to be complete. However, current version of `CLIN-S` requires that semantics with infinite domains be in Bledsoe real arithmetic and non-ground specification is needed.

We omit further details here and more materials can be found in [CP94a].

B.2.2.2 Model Finding

This process mainly focuses on finding a model M for the ground instance set. Since often there are more than one model for a given ground instance set, it is desirable to pick a proper model that can lead to a fast proof. The idea is that a semantic hyper-linking round can be eliminated if we can find, without using semantics, a contradiction between the input clauses and the ground model or its logical consequences. This section is devoted to description of various syntax-based model finding strategies to find a "better" model.

There are two major advantages of using those model finding strategies: Expensive semantic checks can be omitted and since ground logical consequences generated this way introduce no new ground literals, the search space is not expanded while progress can be made.

Essentially the model finding strategies apply inference rules to generate ground logic consequences until M is contradicted by one logic consequence, or the time bound is exceeded (since the inference might not terminate if not bounded). The basic operation is to treat model literals in M as from "unit clauses." Inference rules are applied on those "unit literals" to generate more clauses or literals. When a contradiction occurs, a clause containing negations of the relevant model literals leading to the contradiction, is added. Such new clauses are logical consequences of the input clauses, and the addition of such logic consequences causes the model finding to backtrack to find another model.

The followings are brief description of the model finding strategies used in `CLIN-S`. Further details can be found in [CP93a, CP93b].

Model Filtering When a model M is found for the ground clause set, *filtering* checks if M directly contradicts any input clause. The filtering idea is this: Given a model $M = \{ L_1, L_2, \ldots, L_n \}$, it checks if there is an input clause C and some most general unifier θ such that for each literal L in C, $\neg L\theta \in M$. If such C is found, $C\theta$ (a ground instance) is added to the ground clause set and model finding backtracks to find another ground model (since $C\theta$ contradicts M).

Natural Replacement Model literals in a ground model can be seen as ground units. Such ground units can derive more ground units using *natural replacement rules* which can be automatically generated from the input clauses. The process is called *natural replacement* which generates unit clauses which are logical consequences of the ground model M. For the generated units, a contradiction is checked between any two units. Filtering can also be applied to the new "units." If any contradiction is found, a ground clause containing the *negation* of the relevant model literals (those deriving contradicting units), is added to the ground set. Again, the model finding needs to backtrack to find another model since the added ground clause contradicts the ground model.

Simple Term Rewriting When `equality_predicate/1` is specified (please see Section B.6), a simple kind of term rewriting is applied to the generated units by natural replacement. From the units, the rewrite rules are those positive equality literals oriented in lexicographical order. Input positive equality literals (which might not be ground) are also used as rewrite rules.

With a given rewrite rule, all other units (probably already rewritten earlier) are rewritten to generate new units. A contradiction is checked between any two units. As in natural replacement, a ground clause containing negations of the relevant model literals is added to the ground set. The model finding backtracks to find another ground model.

Minimal Replacement This idea roughly corresponds to *definition expansion* from the ground goal by partially hyper-linking the literals being "defined" with the ground goal. A similar idea has been used by Lee and Plaisted as *predicate replacement* [LP92]. This is helpful to prove some theorems as in set theory that need to use such definitions as $\cup$ or $\cap$.

The idea of minimal replacement is this: Often in a clause corresponding to a definition, all the variables in the definition appear in the literals being defined. For example, the following definition of $\cup$

$$Z \in X \cup Y \equiv (Z \in X \vee Z \in Y)$$

generates the following three clauses:

$$[el(Z,X), el(Z,Y), not(el(Z, union(X,Y)))]. \tag{B.1}$$

$$[el(Z, union(X,Y)), not(el(Z,X))]. \tag{B.2}$$

$$[el(Z, union(X,Y)), not(el(Z,Y))]. \tag{B.3}$$

Once the literals being defined are hyper-linked by ground literals, the definition part will become ground also and a ground clause is obtained. For example, the ground literal *el(a,union(b,c))* could hyper-link with the clause (B.1) and generate the ground clause

$$[el(a,b), el(a,c), not(el(a, union(b,c)))]$$

Further expanding definitions might generate ground clauses needed for the proof.

This idea is generalized to the *minimal replacement rules* generated from the input clauses. In a minimal replacement rule, the left-hand side contain the minimum number of literals such that all the variables on the right-hand side appear on the left-hand side. There might be more than one minimal replacement rule from an input clause (for example, { *not lt(X,Y), not lt(Y,Z), lt(X,Z)* }). Minimal replacement starts with the input ground clauses and hyper-links ground literals with left-hand side of the minimal replacement rules to generate more ground clauses. A proof is found if the generated ground clause set is unsatisfiable.

UR Resolution *UR resolution* is a repeated unit resolution [WCR64] to generate new units. `CLIN-S` applies UR resolution to generate more units from the input units. Newly generated units are retained and incremental model finding does not generate a model that contradicts any generated units from UR resolution.

Note that the units generated by UR, unlike those "units" in natural replacement, are logical consequences of the input clauses and can be retained permanently. Also, UR resolution itself is also a powerful inference rule and helps to find many proofs that would be difficult to obtain without UR resolution.

B.2.2.3 Rough Resolution

In ground instance generation, type 2 literals are generated by enumerating the Herbrand base (see Section B.2.2.1). If some large ground literals need to be generated for the proof, they will not be generated until many other smaller ones are. It might take a long time to generate them. By the time they are generated, the ground clause set might be too large to handle.

Rough resolution tries to cope with this problem by resolving upon potentially large literals so they do not appear in the resolvents. If some large literals are needed for the proof, the instances of resolvents, by resolving upon those large literals, can be used instead.

Rough resolution generates *rough resolvents* in a controlled way. It only resolves upon potentially large literals, *absolute maximal literals* and *likely maximal literals*. Each type of maximal literal is handled separately. Resolvents are retained based on the relative size difference, the *growth*, between the resolvent and the parent clauses. With restrictions on applying resolution and retaining resolvents, rough resolution does not generate as many resolvents (and complicate the search space) as ordinary resolution does.

Semantic hyper-linking applies to rough resolvents as well. However, since sometimes there are still many rough resolvents with the aforementioned restrictions, time bounds have been set so semantic hyper-linking does not spend more time on rough resolvents than input clauses.

Rough resolvents have been shown to help find proofs faster in many cases. For more details about rough resolution, please refer to [CP93c].

B.2.3 Finite Model Finding

`CLIN-S` has a built-in finite model finding procedure implemented by Paramasivam and Plaisted [Pla94, Par] to find a finite model for a satisfiable input clause set. First-order theorem proving is semi-decidable so termination is not even guaranteed for satisfiable input clauses. However, it is known that many decidable sublanguages of first-order logic have the property that if there is a model, there is a model with a finite domain. With this finite model finding procedure, `CLIN-S` can act as a decision procedure for those decidable sublanguages of first-order logic.

Finite model finding, turned on by the flag `use_finite_model_checker` (see Section B.5.1), proceeds in parallel with other `CLIN-S` proving techniques and systematically looks for finite model for the input clause set.

B.3 Using `CLIN-S`

`CLIN-S` has been implemented in Prolog. The current version only runs on Quintus-Prolog [Qui]. Support for other Prolog implementations might be considered in later versions[3].

It is recommended to compile the prover under Quintus Prolog to greatly reduce the time of loading the prover.

The top-level `CLIN-S` prover file is `clins` which in turns loads all other files. A list of complete description of `CLIN-S` source files is given in Section B.9.

The following illustrates the steps to use `CLIN-S` with Quintus Prolog:

- Prepare an file containing the input clauses and `CLIN-S` commands, and another file containing the semantic information. The format of both files is explained in Section B.4 and Section B.6.
- Invoke Quintus-Prolog and the following prompt will show up:

```
| ?-
```

- Read in the prover file by typing (note that the dot should be input as well)

  [clins].

  Several files will be consulted in sequence automatically. Then a welcome message will show up.
- The semantic information file should be read in before the clause file. To load the semantics, type in

[3]Support for C-Prolog [Per84] is difficult because one file of `CLIN-S` (part of the finite model checker by Muthukrishnan Paramasivam) is written in C for better performance. Therefore there is no plan for `CLIN-S` to support C-Prolog.

read_model(Semantics_File).

See Section B.6 for how to specify semantics.

- To start CLIN-S on the input clauses, type in

prove(*Clause_File*).

The following is a typical session of loading CLIN-S to prove file foo with semantics information in file foo_example.

```
Quintus Prolog Release 3.1 (DECstation, Ultrix 4.x)
Copyright (C) 1990, Quintus Corporation.  All rights reserved.
2100 Geng Road, Palo Alto, California U.S.A. (415) 813-3800

| ?- [clins].
% compiling file clins
%  compiling file quintus_2.05.8.a
% Undefined procedures will just fail ('fail' option)
%  quintus_2.05.8.a compiled in module user, 0.200 sec 1,168 bytes
%  compiling file quintus_compile_options_2.05.8

................

%   errno.qof loaded in module date, 0.134 sec 8,800 bytes
%  date.qof loaded, 0.733 sec 23,644 bytes
%  module date imported into user

               WELCOME TO CLIN-S 1.0
(c) 1994 by Heng Chu, Geoff Alexander, Shie-Jue Lee, and David Plaisted



  USER GUIDE :
  Key in "prove(File).".
  Key in "settings." to list all current settings.
```

```
  For more info., type "choice.".
  These info. can be reviewed by typing in "helpmsg.".

% clins compiled in module user, 125.183 sec 738,304 bytes

yes
| ?- read_model(foo_example).

yes.
| ?- prove(foo).
```

B.4 Input Clause File

This section describes the syntax for the input clause file including the input clauses and various CLIN-S commands pertinent to semantic hyper-linking. Specification of semantics (or example) is described in Section B.6.

B.4.1 Comments

Comments can be placed in any file read by Prolog or the prover. They are preceded by the symbol %. All characters from the first % on a line to the end of the line are ignored.

B.4.2 Names

Names are used as constant symbols, function symbols, predicate symbols, propositional constants, individual variables, and commands. The syntactic structure of a name follows the conventions in Prolog. In general, the type (predicate symbol, function symbol, constant) of a name is determined by its context.

The maximum number of characters allowed in a file name is subject to the restriction of the underlying Prolog system.

B.4.3 Input Clause File

The input clause file contains the clauses for the problem to be solved, and consists of commands and lists of clauses. All lists of objects are terminated with "`end.`"

In this section, we describe the specification of clauses. `CLIN-S` commands will be discussed later in Section B.5.

Note that all commands and lists of objects should end with a period ("."). Otherwise it will cause Prolog syntax error.

B.4.3.1 The Structure of an Input Clause File

A typical input clause file to use semantic hyper-linking is structured as follows:

> *commands*
> *axiom list*
> `end_of_input.`
> *commands*

`CLIN-S` starts to solve the input problem when it reads "`end_of_input.`" When done, it executes the commands following "`end_of_input.`" until the end of the file.

All the commands before "`end_of_input`" are executed so as to create the desirable environment for the specific input problem. The commands after "`end_of_input`" are used to restore the normal environment.

Usage of commands will be discussed in detail in Section B.5.

B.4.3.2 Axiom List

The axiom list encodes the axioms and the negation of the conclusion of the input problem. It has the following form:

> `axiom_list(`*Option*`).`
> $axiom_1$
> $axiom_2$
> ...
> $axiom_n$

```
end.
```

Option specifies the form of each $axiom_i$. Two forms are accepted by `CLIN`:

- `clause`: Each axiom is written as a list structure

  $$[L_1,\ L_2,\ \ldots,\ L_k].$$

  Each L_i is a literal. The list is interpreted as

  $$L_1 \vee L_2 \vee \cdots \vee L_k$$

  For example, the axiom

  ```
  [p(X,Z), not p(X,Y), not p(Y,Z)].
  ```

  represents

  $$\texttt{p(X,Z)} \vee \texttt{not p(X,Y)} \vee \texttt{not p(Y,Z)}$$

- `sprfn`: Each axiom is written in the format

  $$L \text{ :- } M_1,\ M_2,\ \ldots,\ M_m.$$

  where L is a positive literal, or `false` if the axiom is negative. L is called the *head* and $M_1,\ M_2,\ \ldots,\ M_m$ consist of the *body* of the axiom. The above expression is interpreted as

  $$\mathrm{L} \vee \neg M_1 \vee \neg M_2 \vee \cdots \vee \neg M_m$$

  For example, the clause in last example can be also represented in `sprfn` format

  ```
  p(X,Z) :- p(X,Y), p(Y,Z).
  ```

A negative literal is represented by `not` followed by an atom. For example, ¬p(X,Y) is represented by `not(p(X,Y))`. The pair of parentheses for `not` is optional.

B.4.3.3 An Example

The following is the input clause file for a problem which says that if for all elements X in a group such that X•X = e, then the group is commutative.

```
assign(time_limit,3000).
axiom_list(sprfn).
p(e,X,X).
p(X,e,X).
p(X,X,e).
p(a,b,c).
false :- p(b,a,c).
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
p(X,V,W) :- p(X,Y,U), p(Y,Z,V), p(U,Z,W).
end.
end_of_input.
```

The first line is a `CLIN-S` command to set the time limit to be 3000 seconds.

B.5 `CLIN-S` Commands

`CLIN-S` provides two kinds of commands. One is to *set/clear* a flag; the other to *assign/delete* a value of a global parameter. Through the execution of commands, `CLIN-S` creates the desired running environment for a given input problem. In this section, we only discuss commands *that are pertinent to semantic hyper-linking*.

B.5.1 Set/Clear Flags

Such commands enable or disable some functions by setting or clearing flags including *control* and *output* flags. A control flag controls `CLIN-S` in its search for proofs; an output flag enables various output during the search of proofs.

To set a flag *Flag*, use

`set(`*Flag*`).`

To clear (reset) a flag *Flag*, use

`clear(`*Flag*`).`

The following is a list of available control set/clear flags provided by `CLIN-S` that are pertinent to semantic hyper-linking.

Flags	*Defaults*	*Note*
use_filtering Section B.2.2.2	set	Enable model literal filtering after model finding found a model for the ground clause set. This function should be always enabled since it is usually fast and finds model contradictions without semantic checks
use_finite_model_checker Section B.2.3	clear	Enable the finite model finding procedure by Paramasivam and Plaisted [Pla94, Par]. Once a finite model is found for the input clauses, `CLIN-S` will print out the model and terminate.
use_minimal_replacement Section B.2.2.2	set	Enable minimal replacement after a ground model is found. Although it is shown to be useful in many cases, minimal replacement might take a long time sometimes without helping the prover.
use_natural_replacement Section B.2.2.2	set	Enable natural replacement after a ground model is found. In general, we have found natural replacement useful and help to find the proofs faster without expensive semantic checks.
use_rough_resolution Section B.2.2.3	set	Enable rough resolution to get rid of large literals. Rough resolution sometimes might generate many rough resolvents (even though restrictions have been set on applying resolution and retaining resolvents) and take a long time.

use_ur_resolution Section B.2.2.2	set	Enable UR resolution. UR resolution has been shown to be a useful inference rule by itself. We often find it helps to find proofs faster and units generated are useful in simplifying other clauses. However, if the number of units grows large, UR might spend much time doing nothing.

B.5.2 Assign/Delete Parameters

Such commands assign or delete a value from a global *parameter*. To assign a value, *Val*, to a parameter, *Parm*, use

`assign(`*Parm, Val*`).`

To delete an entry of a certain parameter, *Parm*, from the database, use

`delete(`*Parm, Arity*`).`

where *Arity* is the arity of the parameter *Parm*. `CLIN-S` stores the parameter and its value in the form

Parm(Val)

All such parameters have arity 1.

Although `CLIN` has many parameters whose values can be adjusted by the user, the following is the list of available parameters for `CLIN-S` that are pertinent to semantic hyper-linking.

Parameters	*Defaults*	*Meaning*
time_limit	2000 seconds	The value can be any real number. `CLIN-S` stops if the time spent for a given problem exceeds the actual time limit which is `time_limit_coef` $\times$ `time_limit`.
time_limit_coef	1	The value can be any real number. The value is used for scaling the specified time limit.

B.6 Semantics Specification

To use semantic hyper-linking, one must specify a semantics (using an example) in the input file. When a semantics is specified, `CLIN-S` automatically uses semantic hyper-linking as the major inference rule. In this section we describe how one can specify the semantics with an example for the theorem to be proved.

A semantics specification is enclosed by `model_start` and `model_end`, and has the following format:

```
start_of_models.
% Specify the function symbols.
semantic_function(Function_list).opt
% Specify the constant symbols.
semantic_constant(Constant_list).opt
% Specify the equality predicate.
equality_predicate(Eq).opt
% Specify at least one model.  See Section B.6.3
Model 1
Model 2 opt
...
Model i opt
end_of_models.
```

Anything after a `%` is treated as a comment. There should be only one semantics specification (enclosed by `start_of_models` and `end_of_models`) for a given problem. A redundant specification (second or more `start_of_models/end_of_models` pairs) will be ignored. An $_{opt}$ indicates an option which needs not to be present. Semantics specifications should be put in a separate file and read in by

`read_model('`*semantics_file*`').`

before the Prolog command

`prove('`*problem_file*`').`

In the future version, it will be allowed to specify the semantics or issue the `read_model` command within the *problem_file*. Specifying semantics separately, however, allows the flexibility to use the same input clause file with different model specifications.

B.6.1 `semantic_function` and `semantic_constant`

Both `semantic_function` and `semantic_constant` are optional since `CLIN-S` can automatically generate them if not given. They are used by `CLIN-S` to generate ground instances. These options are kept to ensure backward compatibility with older versions of `CLIN-S`.

`semantic_function`(*Function_list*) allows one to specify the function symbols in a list. Each function must have its arity specified. This is done by giving correct number of variable arguments (beginning with capital letters). For example, *Function_list* can be `[f(X,Y),g(X)]` to specify that there are only two functions `f/2` and `g/1` in the input.

`semantic_constant`(*Constant_list*) allows one to specify the constant symbols in a list. For example, *Constant_list* can be `[a,b,c]` which specifies that `a`, `b` and `c` are all the constants in the input.

It is advised *not* to specify either `semantic_function` or `semantic_constant` since incorrect specification might make the prover unable to find the proof. However, if the input clauses contain no constants, it is advised to specify one constant using `semantic_constant` if proper semantics needs to be specified as well. Otherwise, although `CLIN-S` automatically generates a constant if the input clauses contain none, the semantic information of this new constant would be unknown to `CLIN-S`.

B.6.2 `equality_predicate`

This option

`equality_predicate`(*Eq*).

specifies the equality predicate symbol *Eq* in the input clauses. This option turns on the simple term rewriting on literals generated by the natural replacement in the model finding. Simple term rewriting is useful to prove problems involving equality. However, `CLIN-S` is not designed to specially handle problems with equality.

`equality_predicate(=)` and `equality_predicate(eq)` are two examples of this option.

B.6.3 Semantics

A semantics contains information on *domain*, *constant interpretations, function interpretations* and *predicate interpretations*. A domain can be *finite* or *infinite*. For example, a semantics of a $\mathbf{Z}_4$ group contains a finite domain; a semantics for a typical problem in analysis has an infinite domain on real numbers. Constant interpretations are needed for ground case; function and predicate interpretations have to be specified in *ground* case for semantic checks on ground literals, and if the domain is infinite, *non-ground* case as well for semantic checks on non-ground literals. A semantics with a finite domain does not need a non-ground interpretation because such task can be achieved using a ground interpretation with variables replaced by all possible values in the domain. For semantics with infinite domains, however, semantic checks on non-ground literals in general need decision procedures.

B.6.3.1 Bledsoe Real Arithmetic

Since non-ground semantic checks require a decision procedure, the choice for infinite semantics unfortunately is not arbitrary because not all have a decision procedure. For example, the whole of real number arithmetic is undecidable. In our prover, we have a built-in decision procedure, *Sup-Inf* [Ble75, Sho77], for infinite semantics expressed by a subset of *Presburger* arithmetic which is called *Bledsoe real arithmetic* in [Bun83].

Bledsoe real arithmetic is a subclass of Presburger arithmetic extended with function symbols *min* and *max*. A formula is said to be in Bledsoe real arithmetic if it is a well formed algebraic expression, allowing:

- universally quantified variables,
- real constants,
- addition and subtraction,
- constant multiplication,

- the arithmetic relations $\leq$ and $=$,
- and the logical connectives such as $\vee, \wedge, \neg, \supset$.

Function symbols *min* and *max* arise during the Sup-Inf procedure, they are not allowed to be used in interpretations. The Sup-Inf procedure determines the validity and invalidity of a given Bledsoe real arithmetic expression over real numbers.

Presburger arithmetic is similar to Bledsoe real arithmetic except that it allows only integer constants, and it allows both universal and existential quantification.

There followings are some examples of expressions in Bledsoe real arithmetic:

$$\forall x \forall y[(x < 3y + 2) \wedge x = 1 \supset x = y]$$
$$\forall x \forall y \forall z[(y + z < 5 \wedge 3z \geq y + 8 \wedge x \leq y) \supset (2 + z > 2y \vee x > z - 5)]$$

Non-ground interpretations for functions and predicates need to be expressed in Bledsoe real arithmetic (see Section B.6.3.2). A literal has to be "expanded" using the function and predicate definition before being submitted for a semantic check.

Conditional Functions Conditional functions such as maximum (or minimum) and piecewise linear functions, can also be specified and later expanded properly to an expression in Bledsoe real arithmetic.

For example, we can define $max(x, y)$ as

$$max(x, y) = \begin{cases} x & \text{if } y \leq x \\ y & \text{otherwise} \end{cases}$$

In first-order formula, we can represent $max(x, y)$ as

$$((y \leq x) \supset max(x, y) = x) \wedge ((x < y) \supset max(x, y) = y)$$

When the function $max(x, y)$ in a formula F is expanded, it is first replaced by a new variable, say v. Suppose the formula after the variable replacement is F'. The definition for $max(x, y)$ is added as a condition to form the expanded formula

$$\underbrace{(((y \leq x) \supset v = x) \wedge ((x < y) \supset v = y))}_{definition\ for\ v=max(x,y)} \supset F'$$

In general for a formula F, we replace piecewise functions by distinct new variables to get F' and then place the conjunction of the definitions as the condition for F':

$$(\bigwedge \textit{piecewise function definitions}) \supset F'$$

The new variables are used in the corresponding function definitions. For example, consider the formula

$$F : max(a, b) + abs(c) < d$$

We can replace $max(a, b)$ by a new variable $v1$, $abs(c)$ by $v2$. We then expand F to obtain the following formula:

$$[\underbrace{(b \leq a \supset v1 = a) \wedge (a < b \supset v1 = b)}_{definition\ for\ v1=max(a,b)} \wedge \underbrace{(0 \leq c \supset v2 = c) \wedge (c < 0 \supset v2 = -c)}_{definition\ for\ v2=abs(c)}] \supset \underbrace{(v1 + v2 < d)}_{F'}$$

The expanded formula can then be checked using the Sup-Inf procedure.

In later sections, we will discuss how to specify (conditional or unconditional) function and predicate interpretation in Bledsoe real arithmetic.

B.6.3.2 Infinite Semantics

A semantics by default has an infinite domain. It must have constant interpretations for the ground case, function and predicate interpretations for both the ground and non-ground cases.

A typical infinite semantics is specified as follows:

```
model_start.
%
% Ground interpretations
%
% n constant interpretations
eval(Const_1, Value_1).
eval(Const_2, Value_2).
...
```

```
eval(Const_n, Value_n).
% m ground function interpretations
eval(Func_1(Arg_1, ...), Value_1) :- Definition_1.
...
eval(Func_m(Arg_1, ...), Value_m) :- Definition_m.
% l ground predicate interpretations
Pred_1(Arg_1, ...) :- Definition_1.
...
Pred_l(Arg_1, ...) :- Definition_l.
%
% Non-ground interpretations
%
% m non-ground function interpretations
function_def(Func_1(Arg_1, ...),[(Value_1, Cond_1), ...])
...
function_def(Func_m(Arg_1, ...),[(Value_1, Cond_1), ...])
% l non-ground predicate interpretations
predicate_def(Pred_1(Arg_1, ...), Definition_1).
...
predicate_def(Pred_l(Arg_1, ...), Definition_l).
model_end.
```

The ordering is not important but *all interpretations must be specified.* In the following we will discuss each in detail.

Constant Interpretations Constant interpretations are specified by

`eval(`$Const_i$, $Value_i$`).`

for each constant $Const_i$. Each $Value_i$ is a value from the domain which need not be specified. Such an interpretation allows retrieval of the constant value by an `eval` call. For example, `eval(a,3)` specifies that constant `a` has the value 3.

Ground Function Interpretations The ground interpretation for a function $Func(Arg_1, \ldots, Arg_n)$ is a *Prolog clause* of the form

$$\texttt{eval(}\mathit{Func(Arg_1,\ \ldots,\ Arg_n)}\texttt{,}\ \mathit{Value}\texttt{)}\ \ \texttt{:-}\ \ \mathit{Definition}\texttt{.}$$

Such an interpretation allows calculation of a function with ground argument, by a proper `eval` call. Note that in *Definition*, *Value* might be assigned a value from the domain. Also, variable names must start with capital letter. There could be other `eval` calls in *Definition*. For example, `eval(`Arg_i, $Value_i$`)` obtains the value for Arg_i. Even if there are multiple semantics specified, each `eval` call corresponds to a definition within the *same* semantics specification.

For an example, consider the function

$$f(x,y) = \frac{x+y}{2}$$

which can be defined as

```
eval(f(X,Y),V) :-
     eval(X,XV),
     eval(Y,YV),
     V is 0.5*(XV+YV).
```

Suppose `eval(a,3)` and `eval(b,7)` are already defined, `eval(f(a,b),V)` will return the value of `V` as 5.0.

Conditional functions like

$$g(x) = \begin{cases} -x & \text{if } -4 \le x \le 0 \\ \frac{x}{2} & \text{if } 0 < x \le 8 \\ x & \text{otherwise} \end{cases}$$

can also be defined as

```
eval(g(X),V) :-
     eval(X,XV),
     ((XV =< 0, XV >= -4) -> V is -XV;
      (XV > 0, XV =< 8) -> V is 0.5*XV;
       V is XV).
```

Ground Predicate Interpretations Ground predicate interpretations allows the evaluation of predicate with ground arguments. They are simply Prolog clauses with predicates as the heads. For a predicate *Pred*(Arg_1, ..., Arg_n), the typical ground interpretation is:

$$Pred(Arg_1, \ldots, Arg_n) \text{ :- } Definition.$$

The result is either success (*true*) or failure (*false*) in Prolog execution. In *Definition*, `eval` calls are usually needed and they refer to corresponding ones defined in the *same* semantics.

For example, the predicate

$$p(x, y) = \begin{cases} true & \text{if } x > y \\ false & \text{otherwise} \end{cases}$$

can be defined as

```
p(X,Y) :-
     eval(X,XV),
     eval(Y,YV),
     % assume '>' is defined on XV and YV
     XV > YV.
```

Suppose `eval(a,3)` and `eval(b,7)` are already defined, `p(a,b)` will fail and `p(b,a)` will succeed.

Sometimes user defined Prolog clauses are needed for `eval` or ground predicate interpretations. In this case, the user-defined Prolog clauses must be defined as ground predicate interpretations within the *same* semantics in which they are used. They can be called with function or predicate ground interpretations.

Non-Ground Function Interpretations Non-ground function interpretations have to be expressed in *Bledsoe real arithmetic* as described in Section B.6.3.1. It has the following format

`function_def(`*Func*(Arg_1, ...), *Value_list*`).`

Value_list contains a list of *(function value, condition)* pairs and have the following format:

$$[(Value_1,[Condition_1]),(Value_2,[Condition_2]),\ldots,(Value_n,[Condition_n])]$$

This corresponds to

$$Func(Arg_1,\ldots) = \begin{cases} Value_1 & \text{if } Condition_1 \\ Value_2 & \text{if } Condition_2 \\ \ldots & \\ Value_n & \text{if } Condition_n \end{cases}$$

Note that each Arg_i must be Prolog variables beginning with capital letters.

$Value_i$: Each $Value_i$ is an expression in Bledsoe real arithmetic without logic connectives or relations, which represents the value of *Func* when $Condition_i$ is true. In addition, each $Value_i$ has the following restrictions:

- Constant multiplications need to be explicitly specified using $c * var$ format, in which c is a constant and var is a variable. The operator $*$ *cannot* be omitted. When c is 1, it can be simply put as var to omit $1*$. For example, $3x + 4y$ should be $3 * x + 4 * y$ and $1 * x + 1 * y$ can be replaced by $x + y$.

- Only addition operation is recognized in `CLIN-S`, the minus ("$-$") operation needs to be expressed by negative constant multiplication with addition. For example, $-x + 2y - 3z$ should be $-1 * x + 2 * y + -3 * z$, and $x - y$ should be $x + -1 * y$.

- Parentheses are allowed for common constant multiplication. For example, $0.5 * (x + y)$ will be properly recognized by `CLIN-S`.

$Condition_i$: Each $Condition_i$, enclosed by square brackets `[` and `]`, contains a list of conjuncted sub-conditions[4]. For example, condition $[c1, c2]$ represents $c1 \wedge c2$.

The logical connectives in each condition are restricted to `#` ($\vee$), `&` ($\wedge$), and `@` ($\neg$). And other logical connectives can be expressed by $\vee, \wedge$, and $\neg$. For example, $p \supset q$ is equivalent to $\neg p \vee q$. The relations are restricted to $=<$ ($\leq$) and $=$. $p < q$ can be

[4]Such a restriction is for backward compatibility. There can be a single sub-condition involving $\wedge, \vee$, and $\neg$.

expressed as $\neg(q \leq p)$. Figure B.2 lists the logical connectives and relations allowed in CLIN-S.

Consider function g on page 183. The non-ground interpretation is

```
function_def(g(X),[(-1*X,[-1=<X, X =< 0]), % −1 ≤ x ≤ 0
                   (0.5*X,[@(X=<0),X=<8]), % 0 < x ≤ 8
                   (X,[@(-1=<X) # @(X=<8)]) % x < −1 ∨ 8 < x
                  ]).
```

For unconditional functions, the condition is empty. For example, the function f on page 183 has the simplest function definition format as follows:

```
function_def(f(X,Y),[(0.5*(X+Y),[ ])]).
```

Non-Ground Predicate Interpretations Non-ground predicate interpretations have to be specified in Bledsoe real arithmetic. It is specified by

predicate_def(*Predicate(Arg*$_1$*, ...)*, *Definition*).

Each Arg_i has to be a Prolog variable beginning with capital letters. As in the function condition described earlier, only the logical connectives and relations listed in Fig. B.2, are allowed in *Definition*.

For example, the predicate p on page 184 has the following non-ground interpretation:

```
predicate_def(p(X,Y),@(Y=<X)).
```

B.6.3.3 Finite Semantics

Finite semantics specification has the following format:

```
model_start.
finite_domain.
domain(Domain_List).
%
% Ground interpretations
```

```
%
% n constant interpretations
eval(Const_1, Value_1).
eval(Const_2, Value_2).
...
eval(Const_n, Value_n).
% m ground function interpretations
eval(Func_1(Arg_1, ...), Value_1) :- Definition_1.
...
eval(Func_m(Arg_1, ...), Value_m) :- Definition_m.
% l ground predicate interpretations
Pred_1(Arg_1, ...) :- Definition_1.
...
Pred_l(Arg_1, ...) :- Definition_l.
model_end.
```

A finite semantics is specified by

`finite_domain.`

and the finite domain is specified by

`domain(`*Domain_List*`).`

Domain_List contains a list of constants which represent all the values in the domain. Those constants do *not have to* be in the input clauses because often the domain has more values than the constants in the input clauses. However, if a constant `a` in *Domain_List* does not appear in the input clauses, the value, *Val*, of `a` must also be specified using `eval(a,` *Val*`)`. An example of finite semantics specification is given in Section B.7.2.2.

A non-ground interpretation is not needed since the validity of non-ground literals can be determined by exhaustively replacing variables with values in the domain.

B.6.3.4 Trivial Semantics

Sometimes the semantics is not available or cannot be specified by Presburger formulas in non-ground interpretations. However, a semantics is still needed for `CLIN-S` to work properly with semantic hyper-linking. In this case, a *trivial* semantics can be specified.

`all_positive_semantics` specifies a trivial semantics in which all positive predicates are true and all negative predicates are false, regardless of the arguments; `all_negative_semantics` specifies the opposite. It is not needed to specify the domain, function and predicate interpretations. Trivial semantics is read in directly by `CLIN-S` like a semantics file:

```
read_model(all_positive_semantics).
```

or

```
read_model(all_negative_semantics).
```

B.6.4 Approximate Semantics

In current version of `CLIN-S`, the *only* infinite semantics allowed is those in Bledsoe real arithmetic. For semantics not expressible in Bledsoe real arithmetic, one can use a semantics expressible in Bledsoe real arithmetic to "simulate" them. Such a simulating semantics is called an *approximate* semantics and would give `CLIN-S` good guidance if it behaves the same on the axiom clauses. Since `CLIN-S` is complete regardless of the input semantics, the use of approximate semantics allows one to take advantage of the *Sup-Inf* procedure. Even with such a limitation, `CLIN-S` performs well on some problems with approximate semantics.

For example, in Fig. B.3 we show the approximate semantics for *division, integer quotient, multiplication*, and *gcd*. Also, for many functions in set theory, we can simulate them using functions on cardinal numbers.

B.6.5 Multiple Semantics

There can be more than one semantics specified to provide a heuristics like human mathematicians often use more than one examples. The resulting semantics is taken as the *intersection* of all specified semantics such that

- a positive literal is true if and only if it is evaluated to be true in all semantics,

- a negative literal $\neg L$ is true if and only if L evaluated to be false,
- a clause $L_1 \vee \ldots \vee L_n$ is true if and only if there is one literal L_i evaluated to be true,

B.7 Examples

B.7.1 Input Clauses

In this section, we give some typical examples for input clauses. These examples allow one to prepare input clauses without consulting with the `CLIN` user's manual.

B.7.1.1 `sprfn` Format

```
% imv
% With continuity literals for interpolation axioms.
%

% The following is a CLIN flag
assign(time_limit,10000).

axiom_list(sprfn).
% facts

p(a, b).
p(f(a), 0).
p(0, f(b)).

% least upper bound axioms.
% lub1

p(X, 1) :- p(X, b), p(f(X), 0).

% lub2
```

```
p(f(g(X)), 0) :- not(p(1, X)).
p(g(X), b) :- not(p(1, X)).
p(1, X) :- p(g(X), X).

% ineqality axioms.

p(X, X).
p(X, Z) :- p(X, Y), p(Y, Z).
p(X, Y) :- not(p(Y, X)).

% interpolation axioms.

p(X, Y) :- p(X, q(Y, X)).
p(X, Y) :- p(q(Y, X), Y).

% continuity axioms.
% f(x) > 0.
p(f(X), 0) :- p(a, X), p(X, b), p(X, h(X)).
p(f(X), 0) :- not(p(Z, h(X))), p(a, X), p(X, b), p(Z, X), p(f(Z), 0).

% f(x) < 0.
p(0, f(X)) :- p(a, X), p(X, b), p(k(X), X).
p(0, f(X)) :- not(p(k(X), Z)), p(a, X), p(X, b), p(X, Z), p(0, f(Z)).

% negation of the theorem.
false :-  p(f(X), 0), p(0, f(X)).
end.

end_of_input.
```

B.7.1.2 clause Format

```
%
% Halting problem input clauses from non-standard clause translation
%

assign(time_limit,6000).

axiom_list(clause).
% Ax: x is an algorithm
% Cx: x is a computer program in some programming language
% Dxyz: x is able to decide whether y halts on given input z
% Pxy: x halts on given input y
% Qxyz: x halts on given input <y,z>
% Oxy: x outputs y
%

[not l53(V1), c(V1)].
[not l53(V1), not p(V1, V1)].
[l53(V1), not c(V1), p(V1, V1)].
[not l54(V2, V1), p(V1, V2)].
[not l54(V2, V1), o(V1, b)].
[not q(V1, V2, V2), not o(V1, b), l52(V2, V1)].
[not l46(V2, V1), not l53(V1), l54(V1, V2)].
[l53(V1), l44(V2, V1)].
[not l52(V1, V2), l44(V2, V1)].
[not l50(V2, V1), p(V1, V2)].
[not l50(V2, V1), o(V1, g)].
[not l49(V1), c(V1)].
[not l49(V1), p(V1, V1)].
[l49(V1), not c(V1), not p(V1, V1)].
[not q(V1, V2, V2), not o(V1, g), l48(V2, V1)].
[not l45(V2, V1), not l49(V1), l50(V1, V2)].
[l49(V1), l43(V2, V1)].
```

```
[not l48(V1, V2), l43(V2, V1)].
[not l42(V2), l45(V2, V1)].
[not l42(V2), l46(V2, V1)].
[not l43(V2, sf1(V2)), not l44(V2, sf1(V2)), l41(V2)].
[not l40, c(sc1)].
[not l40, l42(sc1)].
[not c(V1), not l41(V1), l39].
[not l4, not l39, l40].
[not l37(V1), c(V1)].
[not l37(V1), not p(V1, V1)].
[not p(V1, V2), not o(V1, b), l38(V2, V1)].
[l37(V1), l34(V2, V1)].
[not l38(V1, V2), l34(V2, V1)].
[not p(V1, V2), not o(V1, g), l36(V2, V1)].
[not l35(V1), c(V1)].
[not l35(V1), p(V1, V1)].
[l35(V1), l33(V2, V1)].
[not l36(V1, V2), l33(V2, V1)].
[not l33(V2, sf1(V2)), not l34(V2, sf1(V2)), l32(V2)].
[not c(V1), not l32(V1), l24].
[not c(V1), p(V1, V1), l30(V1)].
[not l31(V2, V1), p(V1, V2)].
[not l31(V2, V1), o(V1, b)].
[not l28(V2, V1), not l30(V1), l31(V1, V2)].
[not c(V1), not p(V1, V1), l29(V1)].
[not l27(V2, V1), not l29(V1), not p(V2, V1)].
[not l26(V2), l27(V2, V1)].
[not l26(V2), l28(V2, V1)].
[not l25, c(sc1)].
[not l25, l26(sc1)].
[not l5, not l24, l25].
[not c(V1), p(V1, V2), l22(V2, V1)].
```

```
[not l23(V3, V2, V1), q(V1, V2, V3)].
[not l23(V3, V2, V1), o(V1, b)].
[not l19(V3, V2, V1), not l22(V2, V1), l23(V2, V1, V3)].
[not l21(V3, V2, V1), q(V1, V2, V3)].
[not l21(V3, V2, V1), o(V1, g)].
[not c(V1), not p(V1, V2), l20(V2, V1)].
[not l18(V3, V2, V1), not l20(V2, V1), l21(V2, V1, V3)].
[not l15(V3), l18(V3, V1, V2)].
[not l15(V3), l19(V3, V1, V2)].
[not d(V2, V3, sf1(V3, V2)), l17(V3, V2)].
[c(sf1(V2)), l16(V2)].
[not l17(sf1(V2), V2), l16(V2)].
[not c(V1), not l16(V1), l14(V1)].
[not l3, not l14(V1), l15(V1)].
[not l13(V3, V2), d(V2, V3, V1)].
[l13(V3, V2), not d(V2, V3, sf1(V3, V2))].
[not l12(V2), not c(V1), l13(V1, V2)].
[l12(V2), c(sf1(V2))].
[l12(V2), not l13(sf1(V2), V2)].
[not l10, c(sc1)].
[not l10, l12(sc1)].
[not a(V1), not l12(V1), l9].
[not l2, not l9, l10].
[not l8(V3, V2), d(V2, V3, V1)].
[not l7(V2), not c(V1), l8(V1, V2)].
[not l6, a(sc1)].
[not l6, l7(sc1)].
[not l1, l2].
[not l1, l3].
[not l1, l4].
[not l1, l5].
[not l1, l6].
```

```
[l1].

end.

end_of_input.
```

B.7.2 Semantics Specifications

In this section, we give three typical examples of problems and their input semantics.

B.7.2.1 Infinite Semantics

```
%
% This is an example (semantics) for IMV (the Intermediate Value
% Theorem)
%
start_of_models.
% The followings will be automatically generated if not given.
% semantic_function([g(X),f(X),h(X),k(X),q(X,Y)]).
% semantic_constant([l,a,b,0]).

model_start.

% constants
eval(l, 0.0).
eval(a, -1.0).
eval(b, 1.0).
eval(0, 0.0).

% functions
eval(g(X), V) :-
        eval(X, XV), V is 0.5*XV.
```

```
eval(f(X), XV) :-
        eval(X, XV).

eval(h(X), V) :-
        eval(X, XV), V is 0.5*XV.

eval(k(X), V) :-
        eval(X, XV), V is 0.5*XV.

eval(q(X, Y), V) :-
        eval(X, XV), eval(Y, YV), V is 0.5*(XV+YV),!.

% predicate
p(X, Y) :-
        eval(X, XV), eval(Y, YV), XV =< YV.

% Non-ground interpretations
function_def(g(X),[(0.5*X,[ ])]).
function_def(f(X),[(X,[ ])]).
function_def(h(X),[(0.5*X,[ ])]).
function_def(k(X),[(0.5*X,[ ])]).
function_def(q(X,Y),[(0.5*(X+Y),[ ])]).

predicate_def(p(X,Y),X=<Y).

model_end.

end_of_models.
```

B.7.2.2 Finite Semantics

```
%
% A semantics (example) for a ring theorem -- for all x, x*0=0
```

```
% where 0 is the additive identity element.
%
start_of_models.

% Optional. They will be generated if not given.
% semantic_function([g(X),f(X,Y),j(X,Y)]).
% semantic_constant([a,k]).

model_start.
finite_domain.

% domain/1 is used when the domain is finite and we can list all of
%       the elements in the domain. semantic_constant contains a
%       subset of domain/1 in this case.
%
domain([k,e,a]).

% A semantics with a finite domain, Z3={0,1,2}=[k,e,a].
%
% The additive operation is j(X,Y), and
% the multiplicative operation is f(X,Y).
% g(X) is the inverse of X.
%
% k (additive identity): 0
% e: 1 (multiplicative identity)
% a: 2
eval(k,0).
eval(e,1).
eval(a,2).

eval(g(X),V) :-                 % inverse
        eval(X,XV),
```

```
        V1 is 3-XV,
        V is V1 mod 3.

eval(f(X,Y),V) :-                % multiplicative operation
        eval(X,XV),
        eval(Y,YV),
        V1 is XV*YV,
        V is V1 mod 3.

eval(j(X,Y),V) :-                % additive operation
        eval(X,XV),
        eval(Y,YV),
        V1 is XV+YV,
        V is V1 mod 3.

% predicate
p(X,Y,Z) :-                      % X*Y=Z
        eval(f(X,Y),V),
        eval(Z,ZV),
        V =:= ZV.

q(X,Y,Z) :-                      % X+Y=Z
        eval(j(X,Y),V),
        eval(Z,ZV),
        V =:= ZV.

r(X,Y) :-                        % X=Y
        eval(X,XV),
        eval(Y,YV),
        XV =:= YV.

model_end.
```

```
end_of_models
```

B.7.2.3 Trivial Semantics

Sometimes it is impossible to specify a semantics, in which case trivial semantics can be used. For example, for the halting problem from non-standard clause translation (Section B.7.1.2), many new predicates, Skolem functions and constants are introduced. Their meanings are obscure. Thus a trivial semantics is convenient.

For trivial semantics in which the true literals are exactly the *positive* literals, one can simply read in the information by

```
read_model(all_positive_semantics).
```

For trivial semantics in which the true literals are exactly the *negative* literals:

```
read_model(all_negative_semantics).
```

B.7.3 Running CLIN-S

In this section we give an example to run CLIN-S. Suppose we want to prove the halting problem, whose input clauses (see Section B.7.1.2) are in file halt_prob, and semantics specification (as described in Section B.7.2.3) in file halt_prob_model.

```
| ?- read_model('halt_prob_model').
| ?- prove('halt_prob').
```

B.8 Output

Currently CLIN-S prints out a lot of information during the the proof process. Although they might be helpful to understand how CLIN-S works, users can safely ignore most of them. In addition to the output explained in CLIN's manual, there are several additional outputs related to semantic hyper-linking. In this section, we describe some information that users might find useful.

B.8.1 Reporting a Proof

When a proof is found, `CLIN-S` will print out a message such as

```
PROOF found by model finding; an unsatisfiable ground set
```

Such messages report that a proof is found and the prover is about to stop. In the following we list all the messages reporting proofs being found by various components of `CLIN-S`:

Semantic Hyper-Linking proof is found if a chosen clause (an input clause or a rough resolvent) is all unit simplified away. This happens when *D literals* are hyper-linked to ground units and *D' literals* are unit simplified by non-ground units. A message

`PROOF FOUND: All simplified away`

or

`PROOF FOUND:` *U, [NU]*

is printed right after the chosen clause for semantic hyper-linking. U and NU are complementary units.

Rough Resolution Like ordinary resolution, a proof is found if an empty rough resolvent is generated. When this is detected, a message

`PROOF FOUND in Rough Resolution:` cl_no_1,[cl_no_2, ..., cl_no_n]

will be printed. All cl_no_i are the clause numbers (*within* the rough resolution) of the parent clauses to derive the empty clause. cl_no_1 is the clause being chosen to resolve upon the large literals.

Model Finding When an unsatisfiable ground set is detected by the model finding procedure, it will print out the following message

```
PROOF found by model finding; an unsatisfiable ground set
```

And an unsatisfiable ground set will automatically be printed near the end of the output, which can be used to verify the proof.

UR Resolution When a unit conflict is detected in UR resolution, a message

`PROOF FOUND (small unit):` *U, [NU]*

or

`PROOF FOUND (BIG unit):` *U, [NU]*

will be printed and a printout of the UR proof immediately follows (please see Section B.8.2). U and NU are complementary units. Users can ignore `small unit` and *BIG unit* since they are concerned with the implementation details of UR resolution.

B.8.2 Printing a Proof

Unfortunately, `CLIN-S` is not yet well equipped to print out a justification for a proof, namely, *how* a proof is found. For example, when an unsatisfiable ground set is found, no information is maintained about which ground clauses are instances of input clauses, rough resolvents or units from UR resolution. Maintaining such information will need major modification of `CLIN-S` and this is part of the future plan for improving `CLIN-S`.

Model Finding When the proof is found by model finding, a set of unsatisfiable relevant clauses will be printed near the end. For example, IMV proof is found by an unsatisfiable ground set and the following message is printed:

```
PC relevant clauses:
          133.  [not p(0,f(l)),not p(f(l),0)]
          139.  [not p(l,h(l)),p(f(l),0)]
          140.  [not p(k(l),l),p(0,f(l))]
          148.  [p(l,h(l)),p(f(l),0)]
          153.  [not p(k(l),q(l,k(l))),p(k(l),l)]
```

```
157.  [not p(q(l,k(l)),l),p(k(l),l)]
167.  [p(0,f(l)),p(k(l),b)]
168.  [not p(f(q(l,k(l))),0),not p(q(l,k(l)),b),
       p(q(l,k(l)),l)]
193.  [not p(0,f(q(l,k(l)))),p(k(l),q(l,k(l))),
       not p(l,q(l,k(l))),p(0,f(l))]
197.  [p(k(l),q(l,k(l))),p(q(l,k(l)),b),
       not p(k(l),b)]
198.  [p(l,q(l,k(l))),p(q(l,k(l)),l)]
199.  [p(0,f(q(l,k(l)))),p(f(q(l,k(l))),0)]
```

Printing such clauses can serve as a verification of the proof found by model finding. If a model is printed out instead, an error has occurred in the model finding and a report should be sent to the authors. This is very useful in finding bugs in `CLIN-S`.

Model Filtering When a model of the ground instance set directly contradicts one input clause, all the contradicted ground instances will be printed (although only the shortest one is added to the ground instance set). The following is a model contradiction found by model filtering for the `EXQ2` problem:

```
>> Model at this moment:
  1.  not r(m,b)
  2.  r(m,m)
  3.  r(b,b)
  4.  not r(b,m)
  5.  r(j,j)
  6.  r(k,k)
  7.  r(b,k)
  8.  r(k,b)
  9.  r(f(b),f(k))
  10.  r(g(b),g(k))
  11.  not r(m,k)
```

```
  12.  p(m,k)
  13.  p(k,j)
  14.  not r(b,j)
  15.  p(j,k)
  16.  not r(m,j)
  17.  not p(m,j)
  18.  not p(j,b)
  19.  not p(k,m)
  20.  not r(j,k)
  21.  not r(j,m)
  22.  r(k,m)
    $$ Input clause contradicted: [r(m,k),not r(k,m)]
    $$ Input clause contradicted: [r(b,m),not r(b,k),not r(k,m)]
    $$ Input clause contradicted: [p(m,j),not r(k,m),not p(k,j)]
    $$ Input clause contradicted: [p(j,b),not r(k,b),not p(j,k)]
    ----------- Contradiction check (Filtering) ends -----------
  > [not r(k,m),r(m,k)]
>> Shortest contradicted clause:
  1.  [r(m,k),not r(k,m)]
```

Natural Replacement When a contradiction is found by natural replacement, the derivation of the contradiction is printed to help verify the correctness. The following is a model contradiction found by natural replacement for the `EXQ2` problem:

```
>> Model at this moment:
  1.  not r(m,b)
  2.  r(m,m)
  3.  r(b,b)
  4.  not r(b,m)
  5.  r(j,j)
  6.  r(k,k)
  7.  r(b,k)
  8.  r(k,b)
```

```
 9.  r(f(b),f(k))
 10.  r(g(b),g(k))
 11.  not r(m,k)
 12.  p(m,k)
 13.  not r(k,m)
 14.  p(k,j)
 15.  not r(b,j)
 16.  p(j,k)
 17.  not r(j,k)
 18.  p(j,b)
 19.  r(j,m)
 20.  r(m,j)
 21.  not p(m,j)
 22.  p(k,m)
 23.  p(b,k)
   ----------- Contradiction check (Filtering) ends -----------
*** Time elapsed in last model finding: 0.734
>> Natural Replacement: 25
   Literal bound on natural replacement: 230
   New literals from replacement: 31
   Contradiction found with derived literal: r(f(k),b)
   ---------- Natural Replacement Derivation ----------
   r(f(k),b) :-
     not r(f(k),k) :-
       not r(k,m) :- []
       p(k,m) :- []
     p(f(k),k) :-
       p(k,m) :- []
       not r(k,m) :- []
     not r(f(k),m) :-
       not r(k,m) :- []
       p(k,m) :- []
   ----------------------------------------------------
   ---------- Natural Replacement Derivation ----------
```

```
  not r(f(k),b) :-
    not r(f(k),k) :-
      not r(k,m) :- []
      p(k,m) :- []
    r(b,k) :- []
  ----------------------------------------------------
 Natural Replacement took: 1.483
 > [not p(k,m),not r(b,k),r(k,m)]
>> Shortest contradicted clause:
  1.  [not r(b,k),r(k,m),not p(k,m)]
```

Minimal Replacement If a model contradiction is found by minimal replacement with an unsatisfiable ground set, the set will be printed out.

UR Resolution When a proof is found by UR resolution, `CLIN-S` attempts to print out the UR proof immediately after reporting the proof being found. However, only the proof steps by UR will be printed. If a unit is not generated by UR, no information will given as to where and how that unit was generated. Each entry in the proof is either a non-unit clause with its clause number (*within* UR resolution) or a unit with its clause number. Duplicated entries will be marked and not be printed.

For example, the following is the UR proof for the problem `ls37`:

```
PROOF FOUND (BIG unit): not p(h(a,e),h(a,e),h(a,e)),
[p(h(a,e),h(a,e),h(a,e))]
   ------------------- UR Proof -----------------------
   For: p(h(a,e),h(a,e),h(a,e)),size(72,[13,13,13])
     NU-> 72,[not t(V,W,X),not t(V,W,Y),not t(V,e,Z),
     p(X,Z,Y)]
     U -> 13,t(V,W,h(V,W)),[]
     U (printed) -> 13
```

```
  U (printed) -> 13
--------------------------------------------------
------------------ UR Proof ----------------------
For: not p(h(a,e),V,V),proof(56,[92,77,92])
  NU.p-> 56,[p(V,W,X),not p(V,Y,Z),not p(Y,V1,W),
  not p(Z,V1,X)]
  U.p -> 92,p(V,g(V),e),[]
  U.p subsumed-> 77,not p(h(a,e),e,e),[]
   By: [new],not p(h(a,e),V,V),proof(56,[92,77,92])
     NU.p-> 56,[p(V,W,X),not p(V,Y,Z),not p(Y,V1,W),
     not p(Z,V1,X)]
     U.p (printed) -> 92
     U.p (printed) -> 77
     U.p (printed) -> 92
  U.p (printed) -> 92
--------------------------------------------------
```

UR proof denoted by `.p` is based on proof size.

B.8.3 Proof Summary

Under

```
Semantic Hyper-Linking Statistics:
```

there are some output messages that users might find useful.

Backtracking by Model Finding This message prints out the number of backtracking (by finding a contracted ground instance) found by the model finding procedure. Four major components, *filtering, natural replacement, minimal replacement,* and *UR resolution*, are listed. For example:

```
Number of model finding backtrackings by:
```

```
Filering:               39
Natural Replacement:    19
Minimal Replacement:    0
UR Resolution:          0
```

Timing Information Timing information for each component of CLIN-S, follows the backtracking count. This is helpful for the users to check if the work by different components are well balanced. For example,

```
Timing Info:
  * Semantic Hyper-Linking..........................: 2273.93
       o Partial Hyper-Linking            : 841.751
            On Input           : 415.318
            On Non-input       : 426.433
       o Ground Instance Generation       : 1432.18
            On Input           : 364.183
            On Non-input       : 1068.0
  * Rough Resolution................................: 1127.95
       o On Absolute                      : 1116.23
       o On Likely                        : 11.717
  * Model Finding...................................: 1371.2
  * Replacement.....................................: 1325.03
       o Natural Replacement              : 1325.03
            On Model Lit Only : 1050.8
            On Units Too       : 274.234
            Simple Rewriting   : 0
       o Minimal Replacement              : 0
            Unbounded          : 0
            Bounded            : 0
  * UR Resolution...................................: 1.499
       o By Total Size                    : 0.932
            On Non-units       : 0.5
```

```
        On Units           : 0.432
   o By Proof Size                     : 0.567
        On Non-units       : 0.3
        On Units           : 0.267
```

B.9 Prolog Source File Descriptions

The prover consists of 17 Prolog files. Their functions are briefly described below.

1. `prover` — This is the top-level file. It shows welcome message and then consults the other 16 files.

2. `auxiliary` — This file contains common CLIN-oriented procedures used in the other files.

3. `clausify` — This file converts a sprfn clause to a general clause represented by a list.

4. `command` — This file lists all CLIN commands, sets default values, contains procedures for setting/resetting options.

5. `exprules` — This file contains expert knowledge to be used by the top-level supervisor.

6. `finfmgen` — This file contains finite model checker implemented by Muthukrishnan Paramasivam.

7. `fls` — This file calculates the ground set Gr(F) to be tested by the propositional calculus prover.

8. `hyper` — This file performs a round of hyper-linking. Instances generated by hyper-link operations are stored in database.

9. `instdel` — This file does instance deletion.

10. `interp` — This file contains the interpreter of the prover. It converts an input file to internal forms. It detects syntactical errors. It also sets various support sets.

11. `library` — This file contains some general-purpose procedures that are used in the prover.

12. `main` — This file is the main file of the prover. It accepts the input file, clean the database, trys the input problem, and outputs the result and statistics.

13. `model_contradiction` — This file contains the code for model finding strategies.

14. `pc` — This file contains the propositional calculus prover which is a refinement of Davis-Putnam's method.

15. `replace` — This file contains procedures for doing predicate replacement. Instances generated from the replacement rules are stored in the database.

16. `semantics_hyper_linking` — This file contains code for semantic hyper-linking at a high level.

17. `semantics_eligible_literal` — This file contains code for model finding (excluding model finding strategies).

18. `semantics_model` — This file contains one implementation of non-ground evaluation using Sup-inf procedure for Bledsoe real arithmetic.

19. `semantics_read_model` — This file contains code for reading in semantics information.

20. `simplify` — This file performs unit simplification, tautology deletion, and duplicate deletion.

21. `spc` — This file does small proof checking.

22. `try` — This file calls hyper-linking, predicate replacement, unit simplification, propositional calculus prover, instance deletion, and small proof checking iteratively. The iteration stops when the problem is detected unsatisfiable or satisfiable, or any overflow is detected.

23. `ur_resolution` — This file contains the code for UR resolution.

24. `xvisor` — This file contains the top-level supervisor.

B.10 Summary

In this section, we list some limitations and unusual features of `CLIN-S` as well as upcoming revisions.

B.10.1 Some Notes of Using `CLIN-S`

1. Since `CLIN-S` is still evolving and undergoes changes steadily, there might be bugs undetected in the released version. If a bug ever occurs, please report to the author immediately with the input clauses and semantics specification.

2. Although semantic hyper-linking is independent of the input semantics, a better semantics usually gives a better performance. If a semantics is not available (for example, the semantics cannot be expressed in Bledsoe real arithmetic or is simply not known), one can always use approximate semantics or a trivial one.

3. Users have to guarantee the consistency of the specified semantics, namely, no literal is specified both true and false.

4. If a proof is reported to be found by model finding with an unsatisfiable ground set, a set of PC relevant clauses should be printed near the end. If a ground model is printed out instead, an error has occurred during model finding procedure. Please report this bug to the authors.

B.10.2 Forthcoming Revisions

The following changes will be included in future revisions of CLIN-S. At this time, they are still restrictions/limitations of CLIN-S.

- Currently the CLIN-S user's manual is a supplement to the CLIN's manual and focuses on using semantic hyper-linking only. Eventually we will combine two manuals into one. At this moment, one has to refer to CLIN's manual to use hyper-linking.
- A flag will be added to turn off the intermediate output CLIN-S prints out during the proving process. Such information often is not important to general users although it might be helpful to understand how CLIN-S works.
- CLIN-S should automatically detect the error when an unsatisfiable ground set is reported by the model finding but later a model is printed out.
- Semantic hyper-linking is still complete if an undecidable infinite semantics is given, namely, one semantics for which no decision procedures exist. In this case, semantic checks on non-ground literals or clauses cannot be carried out and it might be slower to find the proof. However, CLIN-S is still not equipped to accept an undecidable infinite semantics. Currently all infinite semantics must be in Bledsoe real arithmetic.

```
Algorithm Semantic Hyper-Linking(S,$I_0$)
$S$: input clause set
$I_0$: input semantics
begin
  $T = \{\,\}$ % the ground instance set
  $I = I_0$ % current ``model''; initially it is the input semantics
  loop % each loop corresponds to a round
    if $I \models S$
    then
      $S$ is satisfiable
      return
    else
      % Ground instance generation
      $D$ = set of smallest ground instances of clauses in $S$ such that $I \not\models D$
      $T = T \cup D$

      % Model finding for $T$
      if $T$ is unsatisfiable
      then
        $S$ is unsatisfiable % Found a proof
        return
      else
        % Construct a new semantics $I$ from $I_0$ and
        %     the model $M$ for the ground instance set.
        % Use $I$ as a new model and continue the model searching.
        $I$ = a new semantics such that $I \models T$
  forever
end
```

Figure B.1: Semantic Hyper-Linking Algorithm

	Use¡d in `CLIN-S`	Regular Meaning
Logical Connectives	`#`	$\vee$
	`&`	$\wedge$
	`@`	$\neg$
Relations	`=<`	$\leq$
	`=`	$=$

Figure B.2: Logical connectives and relations allowed in `CLIN-S`

Natural Semantics	*Approximate Semantics*
$x\|y$	$x \leq y \wedge x \geq 0$
$gcd(x, y)$	$min(x, y)$
$x * y$	$x + y$
integer quotient of x/y	$x - y, if\ y \leq x; 0\ otherwise$

Figure B.3: An example of using approximate semantics

Bibliography

[AL91] O.L. Astrachan and D.W. Loveland. METEORs: High performance theorem provers using model elimination. In R.S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Kluwer Academic Publishers, 1991.

[And76] Peter B. Andrews. Refutations by matings. *IEEE Trans. Comput.*, C-25(8), August 1976.

[AvE82] K. R. Apt and H. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, July 1982.

[BB82] A. M. Ballantyne and W. W. Bledsoe. On generating and using examples in proof discovery. *Machine Intelligence*, 10:3–39, 1982.

[BH80] W. W. Bledsoe and Larry M. Hines. Variable elimination and chaining in a resolution-based prover of inequalities. In W. Bibel and R. Kowalski, editors, *Proc. of CADE-5*, pages 70–87, Les Arcs, France, 1980.

[Ble75] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *Proc. of the 4^{th} IJCAI*, pages 15–21, Tiblisi, USSR, 1975.

[Ble83] W. W. Bledsoe. Using examples to generate instantiations of set variables. In *Proc. of the 8^{th} IJCAI*, pages 892–901, Karlsruhe, FRG, 1983.

[Ble90] W. W. Bledsoe. Challenge problems in elementary calculus. *J. Automated Reasoning*, 6:341–359, 1990.

[BLM+86] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *J. Automated Reasoning*, 2:287–327, 1986.

[Boy70] R. Boyer. *Locking: A Restriction of Resolution.* PhD thesis, University of Texas at Austin, Austin, TX, 1970.

[Bru91] M. Bruschi. The halting problem. AAR Newsletter, March 1991.

[Bun83] Alan Bundy. *The Computer Modeling of Mathematical Reasoning.* Academic Press, New York, 1983.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[CLP94] Heng Chu, Shie-Jue Lee, and David A. Plaisted. The use of semantics in a Herbrand-based proof procedure. Tutorial of CADE-12, June 1994.

[Coo71] D.C. Cooper. Programs for mechanical program verification. *Machine Intelligence*, 6:43–59, 1971.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

[CP93a] Heng Chu and David A. Plaisted. Model finding in semantically guided instance-based theorem proving. *Foundamenta Informaticae Journal*, 1993. To appear.

[CP93b] Heng Chu and David A. Plaisted. Model finding strategies in semantically guided instance-based theorem proving. In Jan Komorowski and Zbigniew W. Raś, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, pages 19–28, 15–18 June 1993.

[CP93c] Heng Chu and David A. Plaisted. Rough resolution: A refinement of resolution to remove large literals. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 15–20, 11–15 July 1993.

[CP94a] Heng Chu and David A. Plaisted. Semantically guided first-order theorem proving using hyper-linking. In Alan Bundy, editor, *Proc. of CADE-12*, Nancy, France, 1994. To appear.

[CP94b] Heng Chu and David A. Plaisted. The use of Presburger formulas in semantically guided theorem proving. Presented in *The Third International Symposium on Artificial Intelligence and Mathematics*, January 1994.

[CZ92] Ricardo Caferra and Nicolas Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *J. Symbolic Computation*, 13:613–641, 1992.

[Dav63] M. Davis. Eliminating the irrelevant from mechanical proofs. In *Experimental Arithmetic, High Speed Computing and Mathematics*, volume 15, pages 15–30, 1963.

[Der87] Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite system. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier Science Publishers, Amsterdam, 1990.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[Fit90] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, 1990.

[Gel59] H. Gelernter. Realization of a geometry theorem-proving machine. In *Proc. IFIP*, pages 273–282, Paris UNESCO House, 1959.

[Gil60] P. C. Gilmore. A proof method for quantification theory: its justification and realization. *IBM J. Res. Dev.*, pages 28–35, 1960.

[GOBS69] J. Guard, F. Oglesby, J. Bennett, and L. Settle. Semi-automated mathematics. *J. ACM*, 16(1):49–62, January 1969.

[Hen76] Lawrence J. Henschen. Semantic resolution for Horn sets. *IEEE Trans. Comput.*, C-25(8):816–822, August 1976.

[Her74] J. Herbrand. Researches in the theory of demonstration. In J. van Heijenoort, editor, *From Frege to Gödel: a source book in Mathematical Logic, 1879–1931*, pages 525–581. Harvard Univ. Press, 1974.

[Hin92] L. M. Hines. The central variable strategy of Str∔ve. In D. Kapur, editor, *Proc. of CADE-11*, pages 35–49, Saratoga Springs, NY, 1992.

[HW74] L. Henschen and L. Wos. Unit refutations and Horn sets. *J. ACM*, 21(4):590–605, October 1974.

[Kri63] S. Kripke. Semantical analysis of modal logic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kri71] S. Kripke. Semantical considerations on modal logic. In L. Linsky, editor, *Reference and Modality*, pages 63–72. Oxford University Press, 1971.

[LC92] Shie-Jue Lee and Heng Chu. CLIN user's manual. Technical Report TR92–009, Department of Computer Science, University of North Carolina at Chapel Hill, 1992.

[Lee90] Shie-Jue Lee. *CLIN: An Automated Reasoning System Using Clause Linking*. PhD thesis, University of North Carolina at Chapel Hill, 1990.

[Lov78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland Publishing, Amsterdam, 1978.

[LP92] Shie-Jue Lee and David. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *J. Automated Reasoning*, 9:25–42, 1992.

[Luc68] David Luckham. Refinement theorems in resolution theory. In *Proc. of the IRIA Symposium on Automatic Demonstration*, pages 163–190, Versailles, France, December 1968. Also in Lecture Notes in Mathematics, No. 125.

[Luk48] Jan Łukasiewicz. The shortest axiom of the implicational calculus of propositions. In *Proceedings of the Royal Irish Academy*, pages 25–33, April 1948.

[MB88] Rainer Manthey and François Bry. SATCHMO: a theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *Proc. of CADE-9*, pages 415–434, Argonne, IL, 1988.

[McC90] William W. McCune. *OTTER 2.0 Users Guide.* Argonne National Laboratory, Argonne, Illinois, March 1990.

[McC94] William W. McCune. `Otter` *3.0 Reference Manual and Guide.* Argonne National Laboratory, Argonne, Illinois, January 1994.

[MOW76] John D. McCharen, Ross A. Overbeek, and Lawrence A. Wos. Problems and experiments for and with automated theorem-proving programs. *IEEE Transactions on Computers*, C-25(8):773–782, August 1976.

[Nie89] Xumin Nie. *Implementation Techniques in Automatic Theorem Proving.* PhD thesis, University of North Carolina at Chapel Hill, 1989.

[NP90] Xumin Nie and David A. Plaisted. A complete semantic back chaining proof system. In Mark E. Stickel, editor, *Proc. of CADE-10*, pages 16–27, Kaiserslautern, Germany, 1990.

[Opp75] D. Oppen. *A $2^{2^{2^n}}$ upper bound on the complexity of Presburger arithmetic.* PhD thesis, University of Toronto, Toronto, Ontario, Canada, 1975.

[Ost89] J.S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press, 1989.

[PACL92] David. A. Plaisted, Geoffrey D. Alexander, Heng Chu, and Shie-Jue Lee. Conditional term rewriting and first-order theorem proving. In *Proceedings of the Third International Workshop on Conditional Term-Rewriting Systems*, Pont-à-Mousson, France, 8–10 July 1992. Invited Talk.

[Par] Muthukrishnan Paramasivam. Personal communication.

[PC93] David A. Plaisted and Heng Chu. Unguided search with a semantic hyperlinking theorem prover. In *IJCAI 93 Workshop on Automated Theorem Proving*, August 1993.

[Pel86] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *J. Automated Reasoning*, 2:919–216, 1986.

[Per84] F. Pereira. *C-Prolog User's Manual.* SRI International, Menlo Park, California, August 1984.

[Pfe88] Frank Pfenning. Single axioms in the implicational propositional calculus. In E. Lusk and R. Overbeek, editors, *Proc. of CADE-9*, pages 710–713, Argonne, IL, 1988.

[PG86] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symbolic Computation*, 2:293–304, 1986.

[Pla82] David A. Plaisted. A simplified problem reduction format. *Artificial Intelligence*, 18:227–261, 1982.

[Pla84] David A. Plaisted. Using examples, case analysis, and dependency graphs in theorem proving. In Robert Shostak, editor, *Proc. of CADE-7*, pages 356–374, Napa, CA, 1984.

[Pla86] David A. Plaisted. A decision procedure for combinations of propositional temporal logic and other specialized theories. *J. Automated Reasoning*, 2:171–190, 1986.

[Pla88] David A. Plaisted. Non-Horn clause logic programming without contrapositives. *J. Automated Reasoning*, 4:287–325, 1988.

[Pla90] David A. Plaisted. Mechanical theorem proving. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 269–320. Elsevier Science Publishers B. V., 1990.

[Pla94] David A. Plaisted. Finite theorem proving. Unpublished manuscript, 1994.

[Plaar] David A. Plaisted. Equational reasoning and term rewriting systems. In Dov Gabbay and Jorg Siekmann, editors, *The Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1. Oxford University Press, To appear.

[PP91] David A. Plaisted and Richard C. Potter. Term rewriting: Some experimental results. *J. Symbolic Computation*, 11:149–180, 1991.

[PPV60] D. Prawitz, H. Prawitz, and N. Voghera. A mechanical proof procedure and its realization in an electronic computer. *J. ACM*, 7:102–128, 1960.

[Pre29] M. Presburger. Uber die vollstandigkeit eines genissen systems der arithmetic ganzer zahlen. In *Welshem die Addition als einzige Operation hervortritt*, pages 92–101. Sprawozdanie z I Kongresu Matematykow Krajow Slowcanskich Warszawa, 1929.

[Qui] Quintus Computer Systems, Inc. *Quintus Prolog User's Guide.*

[Rei73] Raymond Reiter. A semantically guided deductive system for automatic theorem-proving. In *Proc. of the 3 rd IJCAI*, pages 41–46, Stanford, CA, 1973.

[Rob65a] J. Robinson. Automatic deduction with hyper-resolution. *Int. J. Comput. Math.*, 1:227–234, 1965.

[Rob65b] J. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. *Machine Intelligence*, 4:135–150, 1969.

[San80] David M. Sanford. *Using Sophisticated Models in Resolution Theorem Proving*, volume 90 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *J. ACM*, 24(4):529–543, October 1977.

[SL92] John K. Slaney and Ewing L. Lusk. Finding models: Techniques and applications. Tutorial in CADE-11, 1992.

[Sla67] James R. Slagle. Automatic theorem proving with renamable and semantic resolution. *J. ACM*, 14(4):687–697, October 1967.

[SSY93] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. Technical Report 93/11, Department of Computer Science, James Cook University, Australia, 1993. The library is available by anonymous ftp from `pub/tptp-directory` at `coral.cs.jcu.edu.au` or `flop.informatik.tu-muenchen.de`.

[SWKL74] Fleisig S., Loveland D. W., Smiley A. K., and Yarmush D. L. An implementation of the model elimination proof procedure. *J. ACM*, 21(1):124–139, 1974.

[Tar51] A. Tarski. *A Decision Method for Ordinary Algebra and Geometry*. University of California Press, 1951.

[Wan65] H. Wang. Formalization and automatic theorem-proving. In *Proc. of IFIP Congress 65*, pages 51–58, Washington, D.C., 1965.

[Wan85] Tie Cheng Wang. Designing examples for semantically guided hierarchical deduction. In *Proc. of the 9 th IJCAI*, pages 1201–1207, Los Angeles, CA, 1985.

[WB87] Tie-Cheng Wang and W. W. Bledsoe. Hierarchical deduction. *J. Automated Reasoning*, 3:35–77, 1987.

[WCR64] L. Wos, D. Carson, and G. Robinson. The unit preference strategy in theorem proving. In *Proceedings of the AFIPS Conference 26*, pages 615–621, Washington, D.C., 1964. Spartan Books.

[Win82] S. Winker. Generation and verification of finite models and counterexamples using an automated theorem prover answering two open questions. *J. ACM*, 29:273–284, 1982.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[Wos88] Larry Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[WR73] Larry Wos and George Robinson. Maximal models and refutation completeness: Semidecision procedures in automatic theorem proving. In F. B. Cannonito W. W. Boone and R. C. Lyndon, editors, *Word Problems: Decision Problems and the Burnside Problem in Group Theory*, pages 609–639. North-Holland Publishing Company, 1973.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12:536–541, 1965.

[WRCS67] L. Wos, G. Robinson, D. Carson, and L. Shalla. The concept of demodulation in theorem proving. *J. ACM*, 14:698–709, 1967.

[Wt78] Wu Went-tsun. On the decision problems and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:159–172, 1978.

[WW84] L. Wos and S. Winker. Open questions solved with the assistance of AURA. In W. Bledsoe and D. Loveland, editors, *Automated Theorem Proving: After 25 Years*, pages 5–48. American Mathematical Society, Providence, RI, 1984.