Instance-Based First-Order Methods Using Propositional Calculus Provers

TR97-042
1997

Muthukrishnan Paramasivam

Department of Computer Science
CB #3175, Sitterson Hall
UNC-Chapel Hill
Chapel Hill, NC 27599-3175

Instance-Based First-Order Methods Using Propositional Calculus Provers

by

Muthukrishnan Paramasivam

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

Chapel Hill

1997

Approved by:

Prof. David Plaisted, Adviser

Prof. Purush Iyer, Reader

Prof. Gyula Mago, Reader

Early refutational theorem proving procedures were direct applications of Herbrand's version of the completeness theorem for first-order logic. These instance-based theorem provers created propositional instances of the first-order clauses to be proved unsatisfiable, and tested the instances on a propositional calculus prover. This methodology was not pursued for several decades as it was thought to be too slow. Moreover, the invention of the resolution inference rule changed the direction of theorem proving forever. The success of resolution was largely due to unification. Recently, unification has been incorporated in creating instances of first-order clauses. Furthermore, high-performance propositional calculus provers have been developed in the past few years. As a result, it is possible to realize effective instance-based first-order methods for several applications.

We describe the design of instance-based methods for three different purposes. First, RRTP is a theorem prover based on the idea of replacing predicates with their definitions. We compare its performance with some state-of-the-art theorem provers. Second, we describe a proof procedure for Horn theories. The proof procedure creates instances of the input clauses by backward chaining and reasons forward among the instances to find the proof. Third, we describe the construction of a finite-model finder. Finally, we describe the application of the theorem prover and the model finder on an application--- description logics. We show by empirical means that, contrary to general belief, theorem provers compare well with specialized application-specific techniques for description logics.

MUTHUKRISHNAN PARAMASIVAM. Instance-Based First-Order Methods Using Propositional Calculus Provers
(Under the direction of Professor David A. Plaisted.)

ABSTRACT

Early refutational theorem proving procedures were direct applications of Herbrand's version of the completeness theorem for first-order logic. These *instance-based* theorem provers created propositional *instances* of the first-order clauses to be proved unsatisfiable, and tested the instances on a propositional calculus prover. This methodology was not pursued for several decades as it was thought to be too slow. Moreover, the invention of the *resolution* inference rule changed the direction of theorem proving forever. The success of resolution was largely due to *unification*. Recently, unification has been incorporated in creating instances of first-order clauses. Furthermore, high-performance propositional calculus provers have been developed in the past few years. As a result, it is possible to realize effective instance-based first-order methods for several applications.

We describe the design of instance-based methods for three different purposes. First, RRTP is a theorem prover based on the idea of replacing predicates with their definitions. We compare its performance with some state-of-the-art theorem provers. Second, we describe a proof procedure for Horn theories. The proof procedure creates instances of the input clauses by backward chaining and reasons forward among the instances to find the proof. Third, we describe the construction of a finite-model finder. Finally, we describe the application of the theorem prover and the model finder on an application— *description logics*. We show by empirical means that, contrary to general belief, theorem provers compare well with specialized application-specific techniques for description logics.

Acknowledgements

Some are mathematicians, Some are carpenter's wives...

I would like to thank my adviser David Plaisted for introducing me to theorem proving. He has been an immense source of advice, ideas and patience. Working with him has provided me with a unique perspective of logic and theorem proving that I shall treasure. I would also like to thank my colleagues Yunshan Zhu and Bill Yakowenko for several discussions that cleared many a nagging doubt. Thanks also to Hantao Zhang for the sources of the propositional prover used in this work.

My committee members have been a great source of encouragement. I thank them for taking the time to attend various meetings that I schedules. I always felt better and motivated after a chat with Jim Anderson. I am grateful to Purush Iyer and Don Stanat for discussing several topics related to my work and keeping me on track when David Plaisted was away. And Gyula Mago's help in preparing for the defense was invaluable. Steve Weiss's help in filling in for Don Stanat at the last minute is greatly appreciated.

I would also like to thank David Beard, Sid Chatterjee and James Coggins for finding financial support for me over various summers. Thanks are also due to Janet Jones and Katrina Coble for removing all administrative obstacles in the path to the Ph.D.

What would graduate school have been like but for the company of so many wonderful and interesting people? And how can I name all of them? Their friendship and humor kept me going on many a blue day. I would like to mention a few that I had the good fortune of living with: Srikanth Ramamurthy, Mark Moir, Jon McAllister, Vikki Genys, Subash Krishnankutty, Yunshan Zhu.

And last, but not the least, I would like to thank my family: My parents, my brother Sriram, and my wife Jaisri. I have taken their encouragement and love for granted. I cannot imagine getting anything done without their backing. I dedicate this effort to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Historical Background and Motivation

1.1.1 Mechanizing Reasoning

Automated Theorem Proving (ATP) is concerned with the task of mechanizing mathematical, or logical, reasoning. It is concerned with the mechanical derivation of conclusions from a set of *axioms* by means of *inference rules*[1]. It is concerned with the application of computer programs to perform the task of deriving conclusions using inference rules. It is concerned with applying these computer programs to areas such as verification[Bra92], program synthesis[Gog96], expert systems[Jac89], even solving open conjectures in mathematics[Kol96]. The field of automated reasoning has been around as long as the digital computer; computer programs to prove theorems appeared as early as the late 1950's[Gel59].

Interest in mechanizing reasoning, however, predates the computer by centuries; as early as the seventeenth century Leibniz dreamed of "lingua characteristica" and "calculus ratiocinator" —a universal language and a calculus for reasoning. Following the invention of first-order logic as a formalism in the nineteenth century, and following attempts at formalizing set theory and number theory by Peano and Frege, Hilbert initiated and developed the "formal axiomatic method". Ideally, this methodology would allow the mechanization of proofs with no room for human intuition. For example, all truths and non-truths about the natural numbers could be mechanically detected using the axioms of number theory. This

[1] We assure the reader that these notions are rigorously formalized in the next chapter

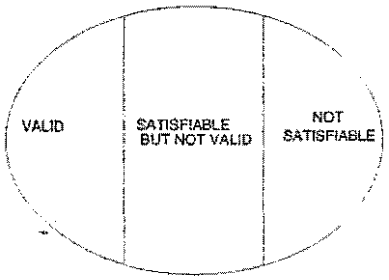

Figure 1.1: The Universe of Well-Formed First-Order Formulas

ideal was first doomed for number theory by Goedel's incompleteness result, and then, by the undecidability result of first-order logic by Turing, for any first-order theory. However, not all hope was lost— the *completeness* of first-order logic established that it was possible to mechanically verify *validity*. For an interesting overview of the history of calculus and mathematical logic that led to these results we refer the reader to [NS93].

1.1.2 Automated Theorem Proving

Consider the universe of first-order sentences as depicted in figure 1.1. Since there is no decision procedure for validity in first-order logic, the boundaries between the regions are not recursively determinable. The scope of automated theorem proving, in the context of first-order logic, is limited to confirming either that a formula is valid, or that it is unsatisfiable. Note that there is an isomorphism between valid formulas and formulas that are not satisfiable: for any valid formula, there exists a formula that is not satisfiable— its negation. Similarly, for every formula that is not satisfiable there exists a valid formula.

The theorem proving problem can be expressed as follows: Given a set of first-order formulas, or axioms of a *theory*, and another first-order formula, or a conjecture, determine whether the conjecture is a *logical consequence* of the theory. If the conjecture does indeed follow from the theory then the conjecture is said to be a *theorem* of the theory.

ATPs solve the theorem proving problem in two different ways: the *affirmative* approach involves systematically computing consequences from the axioms. Highly elegant approaches such as Hilbert Systems[Fit90] and the Sequent Calculus[Fit90] are examples of affirmative theorem proving. The affirmative approach can locate sentences that are present in the region marked as "VALID" in the Figure 1.1. The second approach is the *refutational* method of proving theorems. The refutational method resembles the "proof by

contradiction" mechanism used by mathematicians (and by the rest of us). Refutational methods can locate sentences in the region marked as "NOT SATISFIABLE" in the figure. Since any formula that is valid has an unsatisfiable counterpart, theoretically neither approach is superior to the other. However, researchers have tended to favor refutational systems, sacrificing the elegance of proofs for computational efficiency.

Refutational theorem proving is largely due to Herbrand. Herbrand's important result is that if a first-order formula is unsatisfiable, then there is a computable set of formulas in a less-expressive formalism, the *propositional logic*, that is unsatisfiable. Unlike first-order logic, propositional logic has a sound and complete decision procedure where it is possible to decide whether a propositional formula is valid or not. Note that determining whether a propositional calculus formula is satisfiable is an NP-complete problem.

Early theorem proving procedures[Gil60, PPV60] were direct applications of Herbrand's theorem. Now known as *instance-based* procedures, they consist of two components: a generator of propositional formulas from the first-order formulas, and a propositional calculus decision procedure. These methods did not meet with much success because: *a*) efficient methods to generate the propositional formulas did not exist; *b*) efficient propositional decision procedures were not available. Researchers did little to pursue such methods, as around this time, J.A Robinson revolutionized the field with invention of *Resolution*[Rob65]. With resolution, relatively efficient theorem provers were realized. The resolution inference rule is based on an operation called unification. Unification allows a theorem prover to avoid creating ground instances of the input clauses when searching for a proof.

Resolution has since then conquered the field of symbolic computation as no other technique has. Several semantic and syntactic variants of resolution have been proposed and implemented as theorem provers and other deductive procedures[Apt90]. Donald Loveland lists over twenty-five resolution-based ATP procedures[Lov78]. However, because resolution incorporates unification by combining formulas, usually there is duplication and redundancy in the proof search. Resolution's poor performance in the propositional calculus is well documented[Pla94].

Recently, David A. Plaisted[LP92] developed the clause linking method that combines unification with instance-based methods. Some theorem provers[LP92, Let97] have been built based on this idea and they are comparable to the state of the art provers based on resolution. Moreover, variations of clause linking have been used to incorporate special techniques for equality[Ale95] and semantics[Chu94a, PZ97].

theory[Apt90] and the programming language Prolog [CM81]. The attraction of logic programming is that it is declarative— the denotational semantics of logic programs agrees with their operational semantics. As far as the programmer is concerned, ideally, the execution of such programs is left to methods that maintain the declarative meaning. Although, among programming languages, Prolog exemplifies the declarative paradigm best, efficiency issues have forced several extra-logical constructs into the Prolog programming language.

Resolution-based theorem proving methods thought to be suitable for Horn theories actually perform poorly. These methods either involve too many redundancies in the proof search, or are not specific to the theorem being proved at all. We present a sound and complete technique that selects instances sensitive to the theorem being proved, and searches for the proof by not combining any of the instances, thus avoiding redundancy.

First-order Finite-model Finder

Finding a model for a formula corresponds to locating the formula outside the region marked "NOT SATISFIABLE" in Figure 1.1. Suppose that a first-order formulas is not unsatisfiable. Then, a *model* for the formula may be regarded as a counter-example to the proof of unsatisfiability of the formula. Unlike theorem proving, model finding has received very little attention from researchers. This is primarily because first-order model finding is not even recursively enumerable. However, there are many decidable sublanguages of first-order logic with the *finite-model property*[DG79]. All satisfiable formulas in such sublanguages have finite models.

We present a technique that finds models provided that the formula is satisfiable and has a finite model. Our idea is to encode the model finding problem into a propositional satisfiability problem. The propositional model is then translated back to a first-order model. The idea is very simple and the model finder is extremely quick in uncovering small models.

Description Logics

Finally, we apply our theorem-prover to the domain of description logics, a subset of first-order logic. Description logics, or Concept Languages, are used for representing knowledge using concepts and roles. Description logics have a sound formalism allowing the creation of sound and complete decision procedures. The chief reasoning component

provided in description logic systems is the ability to detect whether some concept is "more general than" (or *subsumes*) another. This is called subsumption checking. The systems also come with the ability of "classifying" the knowledge-base, that is, rapidly checking several subsumptions. Description logics typically have the finite-model property and it seemed apt to use the prover and the model-finder for subsumption checking. Theoretically the theorem-prover-model-finder combination is a complete decision procedure for description logics with the finite-model property. Our effort is completely empirical. We use test cases devised by Heinsohn et al.[HKNP92] to compare several state-of-the-art description logic systems. We report the results of running our tests and also present some propositional techniques for rapid classification.

1.3 Organization of this Dissertation

This thesis is organized as follows: In Chapter 2 we present the preliminaries. The reader may skip this section and come back to it whenever necessary. Chapter 3 describes the construction of RRTP. We describe replacement and how it is used to select instances to create the prover. We also go over the proofs of completeness and soundness of the underlying algorithms, and present some performance results. In Chapter 4 we describe an instance based proof procedure for Horn theories. This proof procedure is compared with some existing theorem provers. Chapter 5 outlines the construction of the model finder. We describe how we encode any first-order formula as a propositional formula and search for its models. In Chapter 6 we compare the performance of the prover and the model-finder with the inferential techniques of some description logic systems. These comparisons are unabashedly empirical. We also describe some new ideas to classify knowledge bases rapidly using propositional methods.

Some of the material in this thesis has been published. Preliminary versions of Chapters 3, 5 and 6 appear in [PP97a], [PP95] and [PP97b].

Chapter 2

Preliminaries

In this chapter we briefly introduce the propositional calculus, first-order logic and logic programming. We also review some specific theorems and techniques in the literature that are relevant to our thesis. This chapter is by no means a complete introduction to logic. For a more thorough presentation the reader is referred to the literature [CL73], [Fit90]. The reader is invited to skip this chapter if and until it is needed. Specifically, the sections on the propositional calculus and first-order logic may be treated as reference material.

2.1 Propositional Calculus

2.1.1 Syntax

A *proposition* is an atomic or an elementary fact that is either *true* or *false* but never both. A declarative sentence such as "Coal is Black" expresses the proposition that coal is black. Another example, "Colorless green ideas sleep furiously" expresses a proposition too. And this may very well be true or false. It does not matter what these sentences mean; their internal structure is irrelevant. As propositions, it only matters whether they are true or false.

We simply represent propositions with letters. For example the letter P, may denote the proposition "Coal is Black", and the letter Q denote the proposition "Colorless green ideas sleep furiously".

Definition 2.1.1 *An atomic proposition is a propositional letter,* T *or* F.

Compound propositions may be constructed from atomic propositions using *logical connectives.* Five logical connectives of interest are: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication, also $\rightarrow$) and $\equiv$ (equivalence). Apart from negation, which is unary, the connectives are all binary. Table 2.1.2 gives an intuitive idea of the semantics of each connective. The semantics are formally introduced in the next subsection. The logical connectives allow the construction of propositions such as "Coal is Black" *and* "Colorless green ideas sleep furiously", or, $P \wedge Q$. The following definitions dictate how the atomic propositions may be combined with logical connectives to create legitimate or *well formed* expressions or *formulas.*

We first define a propositional literal or, simply, a literal.

Definition 2.1.2 *A literal is an atom or the negation of an atom.*

If A is an atom, then the negation of A is written as $\neg A$. Atoms are termed as positive literals and their negations are negative literals.

Definition 2.1.3 *An atom and its negation are termed as complementary literals.*

Definition 2.1.4 *Well-formed formulas in the propositional calculus are defined inductively as follows:*
(1) A literal is a well-formed formula;
(2) If G and H are well-formed formulas. then $G \wedge H$, $G \vee H$, $G \rightarrow H$, and $G \equiv H$ are all well-formed formulas.
(3) All well-formed formulas are created according to the above two rules.

For example, $(P \vee Q) \wedge R$ is a well-formed formula, whereas $P \vee \wedge Q$ is not a well-formed formula.

Definition 2.1.5 *Given well-formed formulas $F_1, F_2, \ldots, F_n$, the formula $F_1 \vee F_2 \vee \ldots F_n$ is the disjunction of $F_1, F_2, \ldots, F_n$; and the formula $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is the conjunction of $F_1, F_2, \ldots, F_n$.*

A clause is a disjunction of literals. All well-formed formulas can be converted to *conjunctive normal form* using simple mathematical properties such as distributivity and De Morgan's laws. This is a normal form that is convenient for theorem proving purposes. There are other normal forms as well.

P	Q	$\neg P$	$(G \wedge H)$	$(G \vee H)$	$(G \supset H)$	$(G \equiv H)$
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

Figure 2.1: Propositional Connectives and their Semantics

Definition 2.1.6 *A well-formed formula that is a conjunction of several disjunctions of literals is said to be in conjunctive normal form. A well-formed formula that is disjunction of several conjunctions of literals is said to be in disjunctive normal form. Formulas in conjunctive normal form are also said to be in clause form. A conjunction of a set of disjunctions of literals is also referred to as a set of clauses. A clause that has only positive literals is called all-positive. A clause that has only negative literals is called all-negative.*

For example, $(P \vee Q) \wedge R$ is in conjunctive normal form. $(P \wedge R) \vee (P \wedge Q)$ is in disjunctive normal form.

2.1.2 Semantics

Classical propositional logic is two-valued. Atomic propositions take truth values that are either *true* or *false*, that is, **T** or **F**. The truth value of an arbitrary well formed formula can be evaluated by evaluating the subformulas it is made of using Table 2.1.2.

Definition 2.1.7 *Given F, a propositional formula, the distinct propositional letters, or the atoms, in the formula make up its atom set.*

The truth-value assignment to the atoms in a formula is made by a function called the interpretation of the formula.

Definition 2.1.8 *Given A, the atom set of a formula F, an interpretation for F is a mapping from A to { **T**,**F** }. An all-positive interpretation maps A to {T}. An all-negative interpretation maps A to {F}.*

Consider the formula, $F_0 = (p \vee q) \wedge (\neg p \vee \neg q)$. The function, $\mathcal{I} : (p \mapsto \mathbf{T}; q \mapsto \mathrm{F})$, is an interpretation of F_0. We note that the formula F_0 maps to T under $\mathcal{I}$.

Definition 2.1.9 *An only positive literals representation of an interpretation, $\mathcal{I}$, for a formula F is a subset of the atom set of F, that $\mathcal{I}$ maps to* **T**.

The only positive literals representation allows for succinct descriptions of interpretations. In the above mentioned example $\mathcal{I}$ would only contain the literal p. In future, we refer to propositional models in their only-positive form.

Definition 2.1.10 *An interpretation M for a formula F, is said to be a model for F, if F evaluates to T under M.*

In the above example, $\mathcal{I}$ is a model for F_0.

Definition 2.1.11 *A formula F is said to be valid, denoted by $\models F$, if it evaluates to* **T** *under all interpretations of F.*

Consider the interpretation function, $\mathcal{I} : (p \mapsto \mathbf{T}; q \mapsto \mathbf{T})$. F_0 maps to **F** under the interpretation $\mathcal{I}$. Since there exists an interpretation under which F_0 is not true, F_0 is not a valid formula. However, the formula $P \vee \neg P$ is a valid formula; no matter how the literal P is interpreted, the formula evaluates to **T**. Valid formulas are also called *tautologies*.

Definition 2.1.12 *A formula F is said to be unsatisfiable if it evaluates to* **F** *under all interpretations of F.*

It is easy to see that if a formula F is valid then $\neg F$ is unsatisfiable.

Lemma 2.1.1 *Every unsatisfiable set S of clauses contains at least one all-positive clause and at least one all-negative clause.*

Proof. If S does not contain any all-positive clauses then an all-negative interpretation models S. If S does not contain any all-negative clauses then an all-positive interpretation models S. □

Lemma 2.1.2 *If a set S of propositional clauses is satisfiable, then there is a model P that contains a literal l from the atom set of S only if l appears as a positive literal in one of the clauses.*

Proof. If S is satisfiable then there is a model M for it. Suppose that there are literals which appear in M but do not appear as positive literals in any of the clauses in S. Then,

construct P by removing all such literals. We know M models every clause in S. Therefore, for any clause in S, either M contains some positive literal that appears in the clause— in which case P contains such a literal as well; or M does not contain some negated literal that appears in the clause— in which case P does not contain it either, because P is a subset of M. Therefore, P models every clause in S. □

Definition 2.1.13 *A propositional formula F is a logical consequence of a propositional formula G, denoted by $G \models F$, if any model for G is a model for F.*

Example. The formula $P \vee Q$ is a logical consequence of P. The formula P is a logical consequence of $P \wedge Q$. This can be verified by inspecting Table 2.1.2. □

2.2 First-Order Logic

In this section we review the syntax and semantics of classical first-order logic. First-Order logic is more general than the propositional logic, it has more expressive power and as a result systematic decision procedures are not to be found. The syntax and semantics of first-order logic are a good deal more complicated than the propositional case. We attempt to be concise but self-contained, however the reader is referred to the literature[Fit90] for a more expository introduction to first-order logic.

2.2.1 Syntax

Definition 2.2.1 *A first-order language is characterized by:*
(1) Boolean Logical connectives.
(2) Quantifiers
(3) Variables
(4) A finite or countable set F of function symbols, each of which has a positive integer associated with it. A function symbol associated with zero is also known as a constant.
(5) A finite or countable set R of predicate symbols, each of which has a positive integer associated with it.

Additionally, languages also have the usual punctuation symbols such as ",","(" ,"(" etc.

Example. $\Sigma = [\{\wedge, \vee, \rightarrow\}, \{\forall, \exists\}, \{x, y, z\}, \{f/1, g/1\}, \{p/2, q/2\}]$ is an example of a first-order language. Example sentences in this language are:
$(\forall x)(\exists y)(p(x, y) \vee q(x, y))$ and
$(\forall x)(\exists y)(p(f(x), y) \wedge q(x, g(y)))$ □

In future we will denote constants by letters from the beginning of the alphabet, and variables from the end of the alphabet. We will use f, g, h to denote function symbols and p, q, r to denote predicate symbols.

Definition 2.2.2 *Terms are defined recursively 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.

Example. Some example terms from Σ are $f(x), g(f(y)), z, f(g(f(y)))$ □

Definition 2.2.3 *The size of a term is defined recursively as follows:*
(1) A constant has size one.
(2) A variable hase size one.
(3) If f is an n-place function symbol, and $t_1, \ldots, t_n$ are terms, then the size of $f(t_1, \ldots, t_n)$ is one added to the sum of the sizes of t_i.

Example. The size of the terms $f(x), g(f(y)), z, f(g(f(y)))$ are two, three, one and four respectively. The size of $h(f(y), g(f(y)))$ is six. □

Definition 2.2.4 *If P is a predicate symbol with arity n and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atom.*

Example. Some atoms from Σ are $p(f(x), g(y)), p(x, z), q(f(x), y)$. □

Definition 2.2.5 *A literal is an atom or the negation of an atom.*

Similar to the propositional logic we have positive and negative literals. An atom is a *positive literal*, and a negated atom is a *negative literal*.

Definition 2.2.6 *Well-formed formulas, or wff's for short, are defined as follows:*
(1) A literal is a wff. All variables that appear in literals are free-variables.
(2) If F and G are wff's and $\circ$ is a binary propositional logical connective then $F \circ G$ is a

wff.

(3) If F is a wff, and x is a free-variable, then $(\forall x)F$ and $(\exists x)F$ are wff. The variable x is now said to be bound.

(4) All wff's are created by a finite number of applications of the above rules.

Example. Some well-formed formulas in Σ are:

$p(x,z) \rightarrow q(z,z)$

$\forall x \forall y(p(x,z) \wedge q(f(x),g(y)))$ □

Definition 2.2.7 *A wff is said to be closed wff if it contains no free variables.*

In the rest of the dissertation we will restrict our attention to well-formed closed formulas.

2.2.2 Semantics

Definition 2.2.8 *A first-order structure $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ of a formula F, consists of a set $\Delta^{\mathcal{I}}$ (the domain of $\mathcal{I}$) and a function $.^{\mathcal{I}}$ (the interpretation function of $\mathcal{I}$) such that:*

(1) Every constant symbol in F is mapped by $.^{\mathcal{I}}$ to an element in $\Delta^{\mathcal{I}}$;

(2) For every f that is an n-place function symbol in F, $f^{\mathcal{I}}$ is a mapping from $\Delta^{\mathcal{I}^n}$ to $\Delta^{\mathcal{I}}$;

*(3) For every p that is an n-place predicate symbol in F, $p^{\mathcal{I}}$ is a mapping from $\Delta^{\mathcal{I}^n}$ to $\{$ **T**, **F** $\}$*

Example. Consider the formula F $\forall x \forall y p(x,y) \wedge q(f(x),g(y))$ created from Σ. We describe $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ an interpretation for F.

$\Delta^{\mathcal{I}} = \{d_1, d_2\}$

$f^{\mathcal{I}} = \{d_1 \mapsto d_2, d_2 \mapsto d_2\}$

$g^{\mathcal{I}} = \{d_1 \mapsto d_2, d_2 \mapsto d_2\}$

$p^{\mathcal{I}} = \{(d_1,d_1) \mapsto \mathbf{T}, (d_1,d_2) \mapsto \mathbf{F}, (d_2,d_1) \mapsto \mathbf{F}, (d_2,d_2) \mapsto \mathbf{F}\}$

$g^{\mathcal{I}} = \{(d_1,d_1) \mapsto \mathbf{F}, (d_1,d_2) \mapsto \mathbf{F}, (d_2,d_1) \mapsto \mathbf{T}, (d_2,d_2) \mapsto \mathbf{F}\}$ □

We sometimes choose to represent the interpretation of a predicate as a set of domain tuples that are mapped to **T**. In the above example we can describe $p^{\mathcal{I}}$ to be the set $\{(d_1,d_2\}$.

From here on all references to first-order clauses will refer to skolemized clauses. All our algorithms and procedures work only on skolemized clauses. Moreover, we will express clauses only as a set of literals. All variables will be assumed to be implicitly quantified. A set of clauses will refer to the conjunction of the clauses. Some clauses in skolem standard form are listed as examples:

Example. $\{p(x,y), q(f(x), g(y))\}$
$\{\neg q(f(x)), p(x,y), r(x,y,z)\}$
$\{\neg q(f(x), y, z), p(x,y), m(f(x))\}$
□

We describe some special kinds of clauses that are of interest to us.

Definition 2.2.22 *A Horn clause is a clause that has at most one positive literal. A unit clause is a clause that has exactly one literal. A ground clause is a clause that has no variables. A clause in which all the clause variables appear in the negative literals of the clause is said to be range-restricted.*

Example. The following clauses are Horn clauses. The first clause is a unit clause and the last one is a ground clause. The third clause is a range-restricted clause.
$\{p(x,y)\}$
$\{\neg q(f(x)), p(x,y)\}$
$\{\neg q(f(x), y, z), p(x,y)\}$
$\{\neg q(a,b), \neg p(a,b)\}$ □

Unification

We now address an important technique used in automated theorem proving. It is the process of making a set of terms or clauses identical by appropriate substitutions. For example, the set of terms $\{g(x, f(a)), g(h(y), z), g(h(b), v)\}$ can be made identical by applying the substitution $\theta = \{x \mapsto h(b), y \mapsto b, z \mapsto f(a), v \mapsto f(a)\}$. But first we need to define what a substitution is.

Definition 2.2.23 *A substitution is a finite set of the form $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$, where every v_i is a variable, every t_i is a term different from v_i, and all the v_i's are distinct. When $t_1, \ldots, t_n$ are ground terms, the substitution is called a ground substitution.*

Example. The following sets are substitutions:

$$\{x \mapsto f(a), y \mapsto f(z)\},\ \{x \mapsto f(g(a)), y \mapsto h(c,d)\}.$$

The second substitution is ground. □

Applying a substitution to a clause creates an instance of the clause.

Definition 2.2.24 *Let C be a clause, and $x_1, \ldots, x_k$ be variables in the clause. Let θ be a substitution of the form $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. The instance of C by θ, denoted by $C\theta$, is the result of simultaneously replacing each variable x_i in C by the term t_i. An instance of clause C by a ground substitution θ, is said to be a ground instance.*

Example. Let C be the clause $\{p(x,y), q(x,y)\}$. Let θ be the substitution $\{x \mapsto f(a), y \mapsto f(z)\}$. Then $\{p(f(a), f(z)), q(f(a), f(z))\}$ is an instance of C by θ. □

Successive substitutions may be combined by the composition operation.

Definition 2.2.25 *Let $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ and $\sigma = \{y_1 \mapsto s_1, \ldots, y_m \mapsto s_m\}$ be two substitutions. Then the composition of θ and σ, denoted by $\theta \circ \sigma$, is the substitution given by the set $\{x_1 \mapsto t_1\sigma, \ldots x_n \mapsto t_n\sigma, y_1 \mapsto s_1, \ldots, y_m \mapsto s_m\}$ and deleting any element $x_i \mapsto t_i\sigma$ if x_i is the same as $t_i\sigma$, and any $y_i \mapsto s_i$ if y_i appears in $\{x_1, \ldots, x_n\}$.*

Example. Let $\theta = \{x \mapsto w, y \mapsto f(z)\}$ and $\sigma = \{x \mapsto t, w \mapsto f(a), z \mapsto g(y)\}$. Then $w\sigma = t$, $f(z)\sigma = f(g(y))$.

Therefore $\theta \circ \sigma = \{x \mapsto t, y \mapsto f(g(y)), w \mapsto f(a), z \mapsto g(y)\}$. □

Now we define unification.

Definition 2.2.26 *Let $S = \{L_1, \ldots, L_n\}$ be a finite set of terms or literals. A substitution θ is a unifier of S if $L_1\theta, \ldots, L_n\theta$ are identical. S is unifiable if there exists a unifier of S.*

Sometimes it may be possible to unify a set of terms by using a number of substitutions. Of these, there is a unique substitution that is more general than the others. We characterize it as follows.

Definition 2.2.27 *Let $S = \{L_1, \ldots, L_n\}$ be a finite set of terms or literals. A substitution θ is a most general unifier of S if for each unifier ϕ of S there exists a substitution η such that $S\phi$ is identical to $S\theta\eta$.*

A large number of algorithms for finding the most general unifier of a set of expressions have been developed. These algorithms, when given a set of terms, either return

the most general unifier or report that the terms are not unifiable. Any theorem proving text [Fit90],[Lov78],[CL73] will have a description of an algorithm to find the most general unifier.

Equality

Equality is considered to be a binary predicate. It is denoted by the infix operator "$=$". The meaning of equality is represented by the following set of formulas which are referred to as the *equality axioms.*

Definition 2.2.28 *The reflexivity axiom is the formula* $\forall x x = x$.

Definition 2.2.29 *Let* f^n *be a function symbol. The substitution axiom for* f^n *is the formula* $\forall x_1, \ldots, \forall x_n \forall y_1, \ldots, \forall y_n (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$.

Definition 2.2.30 *Let* p^n *be a predicate symbol. The substitution axiom for* p^n *is the formula* $\forall x_1, \ldots, \forall x_n \forall y_1, \ldots, \forall y_n (x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \rightarrow f(y_1, \ldots, y_n))$.

2.3 Topics in First-Order Theorem Proving

Refutational Theorem Proving and Herbrand's Theorem

A very important approach to mechanical theorem proving was given by Herbrand. We outline the facts leading to Herbrand's theorem in this section.

Recall that a wff F is unsatisfiable if and only if it is false under all interpretations. Since these may be over several potentially infinite domains, it is inconvenient to consider all of them, Herbrand observed that it was possible to consider just one domain which is made of atoms from F.

Definition 2.3.1 *Let* S *be a set of clauses. The Herbrand Universe* H *of* S *is inductively defined as follows:*
(1) Any constant symbol that appears in S *is a member of* H. *If no constant symbols appear in* S *then* H *contains a single arbitrary constant symbol, say* a.
(2) If f *is an n-place function symbol that appears in* S *and terms* $t_1, \ldots, t_n$ *are members in* H, *then* $f(t_1, \ldots, t_n)$ *is a member of* H.

(3) H is the smallest set containing (1) and (2).

Example. Let S be $\{\{p(x,y)\},\{q(f(a),g(y,y))\}\}$. The Herbrand Universe of S is as follows:
$\{a, f(a), g(a,a), f(f(a)), f(g(a,a)), g(a,f(a)), g(f(a),a), g(f(a),f(a)), g(a,g(a,a))\ldots\}$ □

Definition 2.3.2 *Let S be a set of clauses. The Herbrand Base of S is defined as follows: For every n-place predicate symbol p in S, $p(t_1,\ldots,t_n)$ is in the Herbrand base of S where $t_1,\ldots,t_n$ are terms from the Herbrand Universe of S*

Example. Let S be $\{\{p(x,y)\},\{q(f(a),g(y,y))\}\}$. The Herbrand Base of S is as follows:
$\{p(a,a), q(a,a), p(a,f(a)), p(f(a),a),\ldots\}$ □

Definition 2.3.3 *A Herbrand Interpretation I for a clause set S is defined as follows.*
(1) Every constant is mapped to itself.
(2) Let f be an n-place function symbol and $h_1,\ldots,h_n$ be elements of the Herbrand universe, H, of S. In I, f is assigned a function that maps $(h_1,\ldots,h_n)$, an element in H^n, to $f(h_1,\ldots,h_n)$, an element from the Herbrand universe of S.
(3) Every n-place predicate symbol is assigned a mapping from H^n to $\{\mathbf{T}, \mathbf{F}\}$.

A Herbrand interpretation may be conveniently described as a subset of the atoms in the Herbrand Base that are assigned to T in this interpretation.

Definition 2.3.4 *If a clause set S is true in a Herbrand interpretation I, then I is a Herbrand model of S.*

Recall that a valid wff is true under all interpretations. Herbrand developed an algorithm to find an interpretation that can falsify a given formula. However, if the formula is valid then no such interpretation exists and the algorithm will halt after a finite number of trials. One corollary of this algorithm forms the basis of instance-based theorem proving.

Theorem 2.3.1 *A set S of clauses is unsatisfiable if and only if there is a finite unsatisfiable subset G of ground instances of clauses of S.*

Proof. Proved in [CL73]. □

Resolution

We briefly overview the resolution proof procedure. Resolution[Rob65] is a refutational theorem proving strategy that combines The main operation in resolution is a simply a single inference rule that takes two clauses, and produces a third clause which is a logical consequence of the two clauses. The two clauses are called *parent* clauses and the third clause is called the *resolvent*. We first define the factoring operation.

Definition 2.3.5 *If two or more literals of a clause C have a most general unifier σ, then $C\sigma$ is called a* factor *of C.*

Example. Let C be the clause $\{p(x), p(a), q(x, y)\}$. $\{p(a), q(a, y)\}$ is a factor of C. □

We now define *resolution*. Suppose there are two clauses C_1 and C_2 such that C_1 contains the literal l_1 C_2 contains the literal l_2 such that l_1 and l_2 are complementary by the most most general unifier θ. Then, the resolution of C_1 and C_2 produces the *resolvent*:

$(C_1 \setminus l_1)\theta \cup (C_2 \setminus l_2)\theta$

We refer to factors of the resolvents as resolvents as well.

Example. Consider the following two clauses C_1 and C_2:

$\{p(x, f(y)), \neg q(g(x), z), r(x, z)\}$ and

$\{\neg p(g(u), w), \neg m(u, v), r(v, w)\}$

The literals $p(x, f(y))$ and $p(g(u), w)$, or L_1 and L_2, unify with the most general unifier $\{x \mapsto g(u), w \mapsto f(y)\}$ denoted by θ. The resolvent of the two clauses is:

$\{\neg q(g(g(u)), z), r(g(u), z), \neg m(u, v), r(v, f(y))\}$

Sidenote: The above resolvent has the following factor.

$\{\neg q(g(g(u)), f(y)), r(g(u), f(y)), \neg m(u, g(u))\}$ □

The resolution proof procedure consists of generating resolvents of a set S of clauses, adding these resolvents to S and creating more resolvents, and so on. A clause C is said to be derived by resolution from S if and only if it it can be generated by a sequence of resolutions as described above. We denote this derivation by $S \vdash_R C$. S is unsatisfiable if $\{\}$ can be derived from S by resolution.

Theorem 2.3.2 *The Resolution proof procedure is sound and complete. $S \models \{\}$ if and only if $S \vdash_R \{\}$.*

Proof. For the proof we refer the reader to [CL73]. □

We illustrate the resolution proof procedure with the following example. For the sake of clarity we show only resolvents that are used to derive the empty clause.

Example. Consider the following set of clauses:

1. $\{\neg p(x, g(y)), \neg q(f(y))\}$
2. $\{p(a, z), \neg p(a, f(z))\}$
3. $\{q(f(b))\}$
4. $\{p(x, f(g(b)))\}$

We get

5. $\{\neg q(f(y)), \neg p(a, f(g(y)))\}$ from 1. and 2.
6. $\{\neg p(a, f(g(b)))\}$ from 3. and 5.
7. $\{\}$ from 4. and 6.

Since we get the empty clause the resolution procedure indicates that a proof is found and terminates. □

There are several variants of the binary resolution operations. We list some that are of interest to us. *Unit Resolution* is a resolution operation where at least one of the clauses is a unit clause. *Unit-resulting resolution* (UR resolution) is a multi-step resolution operation on a single clause telescoped into one operation such that final resolvent is a unit clause. These, and several other variants of resolution, are described in detail in [CL73].

The following example helps illustrate the above two variants.

Example. Consider the following set of clauses:

1. $\{\neg p(x, g(y)), \neg q(y), r(x, y)\}$
2. $\{p(a, z)\}$
3. $\{q(g(b))\}$

We get the following resolvent by unit resolution of (1) and (2)

4. $\{\neg q(y), r(a, y)\}$.

We get the following UR-resolvent by UR-resolution of (1) with (2) and (3)

5. $\{r(a, g(b)\}$. □

Instance-Based Proof Procedures

Another refutational approach to theorem proving is the instance-based approach. Instance-based proof procedures are direct applications of Herbrand's theorem. Recall that a set of first-order clauses, S, is unsatisfiable if and only if there exists a finite subset of

ground instances of S, G, such that G is unsatisfiable. A proof procedure that naturally follows from Herbrand's theorem is to systematically generate ground instances of the input clauses and to use a propositional calculus decision procedure to periodically test the ground instances for satisfiability.

Gilmore[Gil60] implemented the first such theorem prover. His prover generated ground instances of the input clauses using the Herbrand base and periodically tested this for satisfiability. The results were predictable— the prover performed poorly. Herbrand bases are generally huge. Therefore, too many instances were created by this prover. Moreover, the propositional calculus procedure used was highly inefficient. For several decades the instance-based methodology was not pursued chiefly because of the above reasons, and because of the overwhelming influence of resolution on ATP research.

Plaisted and Lee [LP92] describe an idea to combine unification with instance-based methods. The prover based on this idea, CLIN, uses the hyperlinking strategy to create ground instances.

Definition 2.3.6 *Suppose S is a set of clauses. Let C be a clause in S of the form, $\{L_1, \ldots, L_n\}$. Suppose there exist literals that appear in S, $M_1, \ldots, M_n$, and a substitution θ, such that the literals $L_i\theta$ and $M_i\theta$ are complementary. Then the clause instance $C\theta$ is said to be a hyper-link instance of C.*

Briefly, the clause-linking proof procedure creates hyper-instances of the input clause set, S, adds them to S and creates more hyper-instances, and so on. Periodically, all the variables in the set of hyper-instances are replaced by some ground term t from the Herbrand universe, and the resulting ground instances are then tested for satisfiability.

Theorem 2.3.3 *[LP92] Clause linking is a complete proof procedure for first-order logic.*

Example. Consider the following set of clauses:

1. $\{p(x, g(y))\}$
2. $\{\neg p(f(x), y), q(h(y))\}$
3. $\{\neg q(z)\}$
4. $\{q(l(x)), r(y)\}$

The following hyper-instances are created and added to the set:

of the replacement paradigm. In RRTP, input clauses are automatically converted into replacement rules. Replacement rules may be derived from input clauses using a variety of ways or replacement strategies.

In clause form representation, the theorem to be proved is a unit clause, and is often ground. Literals in such clauses are already known to be relevant to the proof search, and are called relevant literals. The replacement rules and relevant literals are used to generate instances of the input clauses. We note that in RRTP, replacement rules are used only to select the instances from the input clauses— no explicit replacing of predicate definitions is done. The instances are tested for unsatisfiability by a propositional calculus decision procedure.

The rest of the chapter is organized as follows: First, we formalize replacement and describe some replacement strategies that are used in RRTP. We then describe a version of RRTP that is complete for range-restricted clauses. Following which we describe extending the range-restricted prover to make it complete for first-order logic. Finally we discuss the performance of the prover when compared to other state-of-the-art theorem provers on some difficult theorems.

3.2 Replacement

Consider the clause $C = \{L_1, \ldots, L_k, N_1, \ldots, N_m\}$. A *replacement strategy* creates one or more *replacement rules* of the form $\neg L_1, \ldots, \neg L_k \rightarrow N_1, \ldots, N_m$ from this clause. The commas to the left of $\rightarrow$ represent logical "and" and the commas to the right of $\rightarrow$ represent logical "or". Note that L_i could be negative, that is, $L_i = \neg P_i$. In such cases $\neg L_i$ refers to P_i. If there exist literals, $\neg M_1, \ldots, \neg M_k$ from the set of *relevant literals* and there exists a most general unifier θ such that $L_i\theta = M_i\theta$ for $1 \leqslant i \leqslant k$, then $C\theta$ is a *replacement instance*. We call the literals $L_1\theta, \ldots, L_k\theta$ *antecedent literals* and the literals $N_1\theta, \ldots, N_m\theta$ *consequent literals* of the replacement instance. A variable that is present only in some consequent literal but not in any of the antecedent literals is a *positive replacement variable*. Extending the definition of a range-restricted clause (see definition 2.2.22), we say that a replacement rule is *range-restricted* if it has no positive replacement variables. A replacement strategy is called range-restricted if it creates only range-restricted replacement rules.

RRTP uses three kinds of replacement strategies: *forward*, *definitional*, and *natu-*

ral. Definitional and natural replacement are both refinements of a general range-restricted strategy. The strategy simply is to distribute the literals of a clause such that all the clause variables are present in the literals that appear on the left-hand side of the resulting replacement rule. Such a general strategy creates too many unintuitive replacement rules. However, imposing constraints on the number of literals on each side of the replacement rule leads to more intuitive and useful strategies including definitional and natural replacement.

3.2.1 Definitional Replacement

Suppose there exists a clause $C = \{L, L_1, \ldots, L_k\}$ containing a literal L such that the variables of L are the variables of C. A *definitional replacement* rule obtained from this clause is $\neg L \rightarrow L_1, \ldots, L_k$. Accordingly, if $\neg M$ is a relevant literal, and there exists a substitution θ such that $L\theta = M\theta$, then $C\theta$ is a replacement instance. $L_1\theta, \ldots, L_k\theta$ are added to the set of relevant literals. Definitional replacement can be extended to two literals. If the variables of literals L_1 and L_2 in a clause $C = \{L_1, \ldots, L_k\}$ are the variables of C, then $\neg L_1, \neg L_2 \rightarrow L_3 \ldots L_k$ is a definitional replacement rule. The following example illustrates definitional replacement.

Example. Consider the following clause:
$\{\neg p1(X), \neg p2(Y), p3(Z), \neg p(X, Y, Z)\}$

One definitional replacement rule obtained from this clause is:
$p(X, Y, Z) \rightarrow \neg p1(X), \neg p2(Y), p3(Z)$

Suppose at some point in the proof search we have that $p(a, b, c)$ is a relevant literal, then the clause instance $\{\neg p1(a), \neg p2(b), p3(c), \neg p(a, b, c)\}$
is a definitional replacement instance. The literals $\neg p1(a)$, $\neg p2(b)$ and $p3(c)$ are added to the set of relevant literals. □

3.2.2 Natural Replacement

Suppose there exists a clause $C = \{L, L_1, \ldots, L_k\}$, such that the set of variables of literal L is a subset of the set of variables of $L_1, \ldots, L_n$, then $\neg L_1, \ldots, \neg L_k \rightarrow L$ is a *natural replacement* rule. In many cases, several natural replacement rules may be derived from a single clause. One variant of natural replacement that is more desirable is to turn a clause $\{L_1, \ldots, L_k\}$ to a replacement rule of the form $\neg L_1, \ldots, \neg L_k \rightarrow \perp$. We term this

strategy simple natural replacement.

Example. Consider the following clause:

$\{\neg p1(X), \neg p2(Y), p3(Z), \neg p(X, Y, Z)\}$

There are several natural replacement rules that may be derived from this clause. Since none of the literals have variables unique to itself, there is a natural replacement rule for each literal as a consequent literal. In our system, we refine natural replacement rules to be simple natural replacement rules; where there are no consequent literals, or if the consequent literal in the rule has all the variables in the clause. In this case the natural replacement rule is the contrapositive of the definitional replacement rule from the same clause.

The two natural replacement rules that are create from the above clause are:

$p1(X), p2(Y), \neg p3(Z), p(X, Y, Z) \rightarrow \perp$

$p1(X), p2(Y), \neg p3(Z) \rightarrow \neg p(X, Y, Z)$

The first rule is a simple natural replacement rule. The second is a contrapositive of the definitional replacement rule from the same clause. □

3.2.3 Forward Replacement

Suppose there exists a clause $C = \{\neg L_1, \ldots, \neg L_n, P_1, \ldots, P_j\}$ where the $\neg L_i$'s are negative literals and the P_i's are positive literals. Then, $L_1, \ldots, L_n \rightarrow P_i, \ldots, P_j$ is a *forward replacement* rule.

Unlike the other replacement strategies that depend on the variables in the clauses, forward replacement orients the clause based on the positive and negative literals in the clause. As a result, exactly one forward replacement rule can be derived from every clause. Furthermore, forward replacement is not a range-restricted strategy; it creates rules that have positive variables.

Example. Consider the following clause:

$\{\neg p1(X, Y, Z), \neg p2(X, Z), q1(X, Y), q2(Y, Z)\}$

The forward replacement rule from this clause is

$p1(X, Y, Z), p2(X, Z) \rightarrow q1(X, Y), q2(Y, Z)$

□

3.3 RRTP

RRTP performs several rounds of replacement using rules created by the replacement strategies. We describe how one round of replacement is done and characterize it in terms of its inputs and outputs.

Definition 3.3.1 *Replace takes as input a set of replace rules, $\mathcal{R}$, and two sets of relevant literals, $\mathcal{M}_1$ and $\mathcal{M}_2$. For every replacement rule of the form $P_1, \ldots, P_k \rightarrow N_1, \ldots, N_m \in \mathcal{R}$, suppose there exist literals, $M_1, \ldots, M_k$ in $\mathcal{M}_1 \cup \mathcal{M}_2$, such that $M_1, \ldots, M_k$ and $P_1, \ldots, P_k$ unify with a most general unifier Θ, and at least one of $M_1, \ldots, M_k$ is present in $\mathcal{M}_1$, then, Replace outputs the replacement instance, $P_1\Theta, \ldots, P_k\Theta \rightarrow N_1\Theta, \ldots, N_m\Theta$ and the relevant literals, $N_1\Theta, \ldots, N_m\Theta$.*

That is, $Replace(\mathcal{R}, \mathcal{M}_1, \mathcal{M}_2) =< \mathcal{I}, \mathcal{L} >$, where $\mathcal{I}$ is the set of replacement instances returned and $\mathcal{L}$ is the set of relevant literals created in this round. The reason for dividing the input relevant literals into two sets is to avoid recreating the same replacement instance in every round. This will become clear during the algorithm's discussion.

3.3.1 A Prover for Range-Restricted Clauses

We first describe a prover for range-restricted clauses. Replacement rules created from these clauses using definitional, natural and forward replacement are range-restricted. The reason for describing this version of the prover is twofold: completeness of RRTP can be demonstrated more easily by first showing that this prover is complete on range-restricted clauses. Secondly, it is not uncommon to find theorems that involve only range-restricted clauses, and most theorems have many range-restricted clauses, including ground positive clauses.

Figure 3.1 shows the prover for range-restricted clauses. The prover repeatedly performs rounds of replacement. In each round the prover uses the relevant literals created so far to produce replacement instances, to create more relevant literals and ground instances. The ground instances are tested for unsatisfiability every round.

Completeness

We discuss the completeness of this prover for range-restricted clauses. From Herbrand's theorem, if a set of clauses $\mathcal{S}$ is unsatisfiable then there is a set of ground instances

Algorithm 1 (RRTP, Range-Restricted Version)
Input: Set $\mathcal{S}$ of skolemized range-restricted first-order clauses
Output: Proof of Unsatisfiability of $\mathcal{S}$.

```
1   Round ← 1;
2   PROOF ← false;
3   SATISFIABLE ← false;
4   ℒ ← φ;
5   L_0 ← φ;
6   ℛ ← Replacement rules from 𝒮 using replacement strategies;
7
8   while not PROOF and not SATISFIABLE do
9      i ← Round;
10     < S_i, L_i > ← replace(ℛ, L_{i-1}, ℒ);
11     ℒ ← ℒ ∪ L_i;
12
13     𝒮 ← 𝒮 ∪ S_i;
14
15     if 𝒮 is PCunsatisfiable then
16        PROOF ← true;
17     endif
18     if L_i is φ then
19        SATISFIABLE ← true;
20     endif
21
22  endwhile
```

Figure 3.1: RRTP for range-restricted clauses

of S that is propositionally unsatisfiable. Let the minimal unsatisfiable subset of the ground instances be G_S. It suffices to show that the replacement instances created by the prover will eventually contain the minimal unsatisfiable set. We need only forward replacement to show completeness, and we assume that the prover uses only forward replacement.

The proof is along the lines of the following argument: Every round of replacement creates at least one new ground instance that is part of the minimal unsatisfiable set. Since the number of ground instances in G_S is bounded, the prover eventually generates all ground instances in the minimal unsatisfiable set.

Lemma 3.3.1 *Suppose there exists a satisfiable set of replacement instances, then there is a model for the replacement instances, that contains only relevant literals.*

Proof. From Lemma 2.1.2, there is a model that contains only positive literals of the clauses. From the definition of forward replacement, all the positive literals in the replacement instances are relevant literals. □

We refer to such a model as the *relevant literal model.*

Lemma 3.3.2 *Given an unsatisfiable set of range-restricted clauses, S, as long as the replacement instances created thus far are satisfiable, each new round of RRTP creates a new replacement instance belonging to the set of minimal unsatisfiable set, G_S, of ground instances of S.*

Proof. The proof is by induction. The first round of replacement produces all the all-positive ground clauses as replacement instances. By Lemma 2.1.1, at least one of them belongs to G_S. Therefore the basis holds.

Suppose that for k rounds the prover creates one new replacement instance that belongs to G_S. If the replacement instances are unsatisfiable then we are done. Otherwise, there exists at least one ground instance, $C\theta \in G_S$ that is, following Lemma 3.3.1, contradicted by the relevant literal model M_R. Therefore, for each negative literal $\neg L\theta$ in $C\theta$, there exists a relevant literal $L\theta$ in M_R. Following the definition of forward replacement, the replacement instance $C\theta$ is created in the $k + 1$th replacement round. □

Theorem 3.3.1 *RRTP is complete for range-restricted clauses.*

Proof. Given a set S of clauses, from Herbrand's theorem, there exists a finite subset of ground instances of S that is unsatisfiable, called G_S. From Lemma 3.3.2, every replacement

round of the prover creates at least one new instance in the unsatisfiable set of ground instances of S. Since the number of instances is bounded, the prover eventually creates all of G_S. □

We illustrate the working of the prover with a simple example.

Example. Consider the following set of range-restricted clauses:

1. $\{largest(l)\}$
2. $\{smallest(m)\}$
3. $\{inRange(x,y,z), larger(x,y), smaller(x,z)\}$
4. $\{\neg largest(x), smaller(z,x)\}$
5. $\{\neg smallest(x), larger(z,x)\}$
6. $\{\neg larger(x,y), smaller(y,x)\}$
7. $\{\neg smaller(x,y), larger(y,x)\}$
8. $\{\neg larger(x,y), \neg larger(y,x)\}$
9. $\{\neg smaller(x,y), \neg smaller(y,x)\}$
10. $\{\neg inRange(a,l,m)\}$

The above clauses define what it means for an object to be in the range of two other objects. The also provide partial definitions for relations between objects in terms of size. The theorem states that any object lies in the range between the smallest and largest object.

Some replacement rules that are derived from the above set of clauses are as follows:

1. $\rightarrow largest(l)$
2. $\rightarrow smallest(m)$
3. $\neg inRange(x,y,z) \rightarrow larger(x,y), smaller(y,z)$
4. $\neg largest(x), \neg smaller(z,x) \rightarrow \bot$
5. $\neg smallest(x), \neg larger(z,x) \rightarrow \bot$
6. $larger(x,y) \rightarrow smaller(y,x)$
7. $smaller(x,y) \rightarrow larger(y,x)$
8. $larger(x,y) \rightarrow \neg larger(y,x)$
9. $smaller(x,y) \rightarrow \neg smaller(y,x)$
10. $\rightarrow \neg inRange(a,l,m)$

Rules 1,2,6 and 7 are forward replacement rules from clauses 1,2,6 and 7 respectively. Rules 3,8,9 and 10 are definitional replacement rules from clauses 3,8,9 and 10 respectively. Rules 4 and 5 are natural replacement rules from clauses 4 and 5.

In the first round, from rules 1,2 and 10 we get the following replacement instances:

1. $\{\underline{largest(l)}\}$ 2. $\{\underline{smallest(m)}\}$ 3. $\{\underline{\neg inRange(a,l,m)}\}$
The relevant literals created in this round are underlined. The next round produces the following replacement instance by rule 2:
4. $\{inRange(a,l,m), \underline{larger(a,l)}, \underline{smaller(a,m)}\}$

Following this, we get from rules 6 and 7:
5. $\{\neg larger(a,l), \underline{smaller(l,a)}\}$
6. $\{\neg smaller(a,m), \underline{larger(m,a)}\}$
In the next round we get, from rule 8 and 9:
7. $\{\neg larger(m,a), \underline{\neg larger(a,m)}\}$
8. $\{\neg smaller(l,a), \underline{\neg smaller(a,l)}\}$

Finally we get the following instances from rules 4 and 5:
9. $\{\neg largest(l), smaller(a,l)\}$
10. $\{\neg smallest(m), larger(a,m)\}$
The above instances are propositionally unsatisfiable. □

3.3.2 Instantiating Replacement Instances

To make RRTP complete for full first-order logic, positive variables in the replacement rules that are not range-restricted have to be systematically instantiated by terms from the Herbrand universe. One way to do this is to modify the replacement rules themselves and make them range-restricted by the introduction of *domain* predicates. This is similar to the prover SATCHMO[MB88].

Suppose we have a set S of first-order clauses. For every clause C in S we perform the following modification, which is based on the forward replacement rule created from C. For every positive variable, X, that appears in a replacement rule, a literal *domain(X)* is added to the list of antecedent literals. For example, the replacement rule
$p(x,y), q(y,z) \rightarrow r(x,y,z), t(x,w)$
is modified to
$p(x,y), q(y,z), domain(w) \rightarrow r(x,y,z), t(x,w)$
We refer to the modified clause as C_{domain}, and the modified set of clauses as S_{domain}. The following replacement rules are added to S_{domain} as well: For every constant symbol c, that

appears in S a replacement rule

$\rightarrow domain(c)$

is added. If there are no constant symbols in the clause set, then an arbitrary constant symbol is chosen. For every n-place function symbol f that appears in the clause set the replacement rule

$domain(x_1), \ldots, domain(x_n) \rightarrow domain(f(x_1, \ldots, x_n))$

is added, where $x_1, \ldots, x_n$ refer to universally quantified variables.

The rationale for the above replacement rules is obvious: For every term t in the Herbrand Universe of terms, these forward replacement systematically creates a relevant literal of the form $domain(t)$. In the first round of replacement, from these rules, all the constants in the universe appear in the relevant *domain* predicate literals. Subsequent replacement rounds produce *domain* predicate literals that contain terms from the Herbrand Universe of S.

Lemma 3.3.3 *Any model for S_{domain} must interpret the domain predicates as true.*

Proof. The proof is by induction on the size of the term in the domain predicate. □

Theorem 3.3.2 *S_{Domain} is satisfiable iff S is satisfiable.*

Proof. (If) Suppose S_{domain} is satisfiable and is modeled by M_{domain}. Construct M from M_{domain} by removing exactly the interpretation for the *domain* predicate. Consider any clause $C \in S$. The clause $C_{domain} \in S_{domain}$ contains all the literals in C and, possibly, some negative *domain* literals. By lemma 3.3.3 any model of S_{domain} interprets all domain predicates as *true*. Therefore, if M_{domain} models C_{domain}, then M models C.

(Only If) Suppose S is satisfiable and is modeled by M. Construct M_{domain} from M by adding the interpretation of the *domain* predicate; all *domain* predicates are interpreted to be **T**. Therefore, M_{domain} models all the Horn clauses describing the *domain* predicates. For the other clauses in S_{domain}, suppose M_{domain} contradicts C_{domain}, then since M is identical to M_{domain} except for the *domain* literals, M must contradict C. This is not possible as $C \in S$ and M models S. Therefore M_{domain} models S_{domain}. □

Corollary 3.3.3 *Suppose S_{domain} is unsatisfiable, then by Herbrand's theorem there exists a finite set of ground instances G_{domain} that are unsatisfiable. Let G be a subset of the*

each I_i, bounding the number of replacement rounds for each I_i compromises the prover's completeness.

We desire a version that is complete and prefers small replacement instances. To accomplish this we postpone the instantiation of the variables until after the replacement instances are created. The relevant literals are now possibly non-ground, and have to be unified with the antecedent literals of the replacement rules; previously they only needed to be matched. This results in replacement instances that may have non-positive variables that need to be ground. This strategy allows us to choose an instantiation mechanism that prefers small instances in terms of the substitution size for the positive variables, and keeps the prover complete.

Before we describe the algorithm, we describe an instantiation scheme that is based on the size of the substitution and the number of variables to be substituted. The exact substitution used on an instance depends on the round the replacement instance was created, and the round of the algorithm currently taking place. We define *herbrandTerms* that creates substitutions given the number of variables, the function symbols to create the Herbrand terms and the size of the substitution.

Definition 3.3.2 herbrandTerms *takes as input a set, $\mathcal{F}$, of function and constant terms; a size, N; and the number of variables in the substitution, and outputs all substitutions of size N, of the variables, with substituands made of Herbrand terms created from $\mathcal{F}$.*

The size of a substitution is the sum of the sizes of the substituand terms. For example the substitution $\{x \mapsto f(c), y \mapsto f(g(a), b)\}$ has a size of three, whereas the substitution $\{x \mapsto f(c)\}$ has a size of one.

That is, $\Theta \leftarrow herbrandTerms(\mathcal{F}, N, NumVar)$ where Θ is the set of substitutions created.

The following example illustrates *herbrandTerms*:

Example. Let $\mathcal{F}$ be $\{f(\$, \$), g(\$), a\}$. Let the size of the substitution be one, and the number of variables be two: x_1 and x_2. Then, $herbrandTerms(\mathcal{F}, 1, 2)$ returns the following substitutions:

$\{x_1 \mapsto a, x_2 \mapsto g(a)\}$
$\{x_1 \mapsto g(a), x_2 \mapsto a\}$

$\{x_1 \mapsto a, x_2 \mapsto f(a,a)\}$
$\{x_1 \mapsto f(a,a), x_2 \mapsto a\}$

3.3.3 The RRTP Algorithm

The algorithm that describes RRTP is shown in Figure 3.2. Using the replacement rules created from the input clauses, and the relevant literals computed thus far, every round of replacement produces replacement instances and some new relevant literals. The input relevant literals are divided into two groups to avoid creating the same replacement instance each round.

Every round also creates substitutions, increasing in size, with the Herbrand terms of the input clauses. The replacement instances are made ground with the substitutions to create ground instances. Every round, the ground instances created thus far are tested for unsatisfiability by a propositional calculus decision procedure.

3.3.4 Completeness and Soundness

Theorem 3.3.5 *RRTP is a sound proof procedure for first-order logic*

Proof. The soundness of RRTP is easily established. if RRTP indicates the clause set S is unsatisfiable, then there exists a set of ground instances of S that are propositionally unsatisfiable. Therefore, by Herbrand's theorem, S is unsatisfiable. □

We use the Lemma 3.3.4 to prove the completeness of RRTP.

Lemma 3.3.4 *For every replacement instance $C_{domain}\theta$ created by the range-restricted prover, RRTP creates a ground instance $C\phi$ such that $C\phi$ is the same as $C\theta$.*

Proof. Follows from Definition 3.3.1 and Definition 3.3.2 □

Theorem 3.3.6 *RRTP is refutationally complete for first-order logic.*

Proof. Let S be an unsatisfiable set of first-order clauses. By Theorem 3.3.4 the range restricted version of RRTP creates an unsatisfiable set of ground instances from S_{domain}. From Lemma 3.3.4 RRTP creates a set of ground instances of S that are, by Corollary 3.3.3, unsatisfiable. □

Algorithm 2 (RRTP, Basic Description)
Input: Set $\mathcal{S}$ of skolemized first-order clauses
Output: Proof of Unsatisfiability of $\mathcal{S}$.

```
1   Round ← 1;
2   PROOF ← false;
3   SATISFIABLE ← false;
4   ℒ ← φ;
5   L_0 ← φ;
6   ℛ ← Replacement rules from S using replacement strategies;
7
8   while not PROOF and not SATISFIABLE do
9      i ← Round;
10     S_i ← φ;
11     < R_i, L_i > ← replace(ℛ, L_{i-1}, ℒ);
12     ℒ ← L ∪ L_i;
13
14     for j = 0 to i do
15        k ← i − j;
16        forall I ∈ R_j do
17           forall Θ ∈ herbrandTerms(ℱ, NumVars(I), k); do
18              S_i ← S_i ∪ I.Θ;
19           enddo
20        enddo
21     enddo
22
23     S ← S ∪ S_i;
24
25     if S is PCunsatisfiable then
26        PROOF ← true;
27     endif
28     if L_i is φ then
29        SATISFIABLE ← true;
30     endif
31
32  endwhile
```

Figure 3.2: Outline of RRTP Algorithm

3.3.5 UR Resolution

UR-Resolution combines well with replacement. A set of unit consequences $\mathcal{U}$ of the input clauses is maintained, starting with the unit clauses from the input. Suppose that there is a natural replacement rule of the form $A_1, \ldots, A_n \rightarrow L$. If there exist $U_1, \ldots, U_n$ in $\mathcal{U}$, and a substitution θ such that $A_i\theta = U_i\theta$, then $L\theta$ is added to $\mathcal{U}$. Unsatisfiability is detected by the creation of a replacement instance $\perp$ from a replacement rule of the form $A_1 \ldots A_n \rightarrow \perp$.

Unit consequences are treated like relevant literals when creating replacement instances, except the literal in the replacement rule that unifies with a unit consequence is resolved away in the replacement instance. For example, suppose that there is a replacement rule, $L_1, \ldots, L_k \rightarrow A_1, \ldots, A_j$, and there exist relevant literals, $M_1, \ldots, M_m$, such that $L_i\theta = M_i\theta$, $1 \leqslant i \leqslant m$, and $L_{m+i}\theta = U_i\theta$, $1 \leqslant i \leqslant k - m$, where $U_1, \ldots, U_{k-m}$ belong to the set $\mathcal{U}$. Then, the replacement instance $\{ \neg L_1\theta, \ldots, \neg L_m\theta, A_1\theta, \ldots, A_j\theta \}$ is created. If the replacement instance is made of a single consequent literal, then that literal is added to $\mathcal{U}$.

Example. Consider the following replacement rule:

$p1(X, Y, Z), p2(X, Z) \rightarrow q1(X, Y), q2(Y, Z)$

Suppose we have the relevant literal $p1(a, b, c)$ and the literal $p2(a, c)$ is a unit consequence literal. Then, the following replacement instance is created.

$p1(a, b, c) \rightarrow q1(a, b), q2(a, c)$

The literals $q1(a, b)$ and $q2(a, c)$ are added to the set of relevant literals. □

Example. Consider the following replacement rule:

$p1(X, Y), p2(X, Z) \rightarrow q1(X, Y, Z)$

Suppose we have the unit consequence literals $p1(a, b)$ and $p2(b, c)$. Then, the following replacement instance is created:

$\{q1(a, b, c)\}$

The literal $q1(a, b, c)$ is then added to $\mathcal{U}$. □

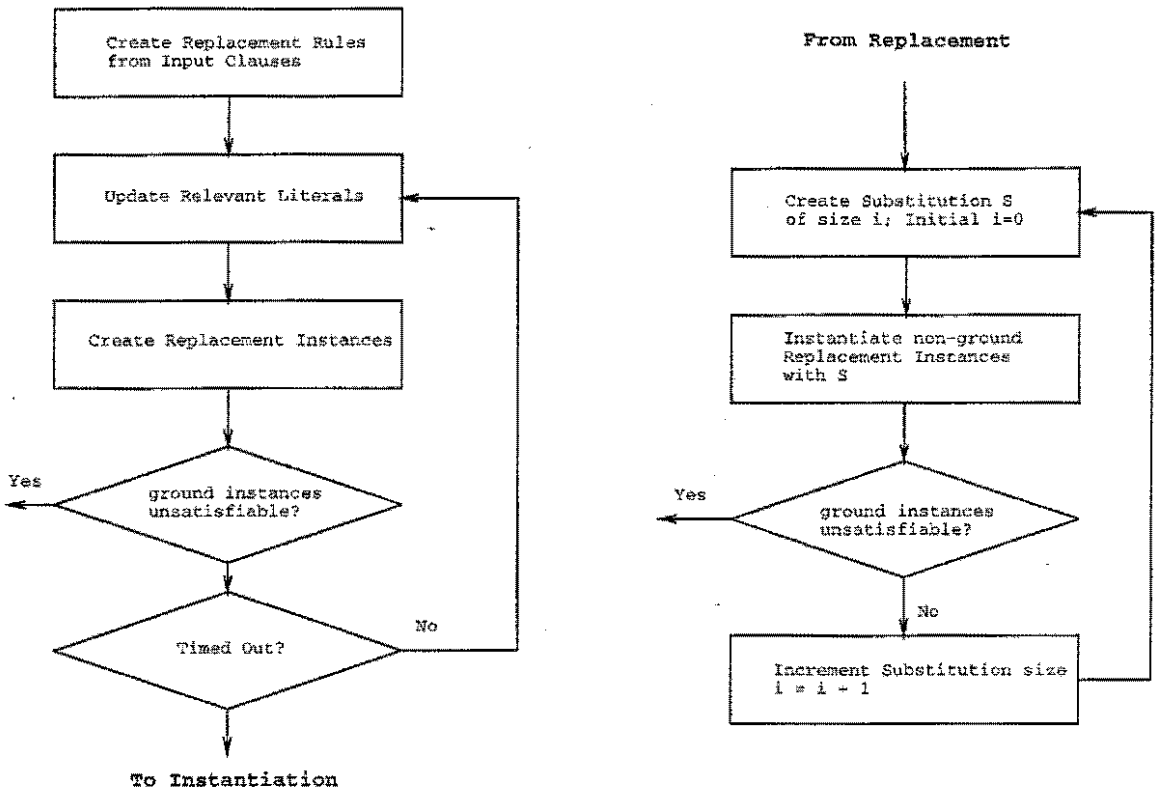

Figure 3.3: The chart on the left depicts the Replacement phase, while the one on the right depicts the Instantiation phase.

3.3.6 Improvements and Variants

Ground Replacement

In many cases, the theorem is represented by ground unit clauses in the input set. Clearly, these literals are relevant to the proof. Moreover, any positive clause that is relevant to the proof is likely to be ground. Other positive clauses are usually unit clauses and such clauses are not used as replacement rules. The prover can be modified to use relevant literals from the ground unit clauses and only range-restricted replacement rules, to create replacement instances that are all ground. Of course, such a modification destroys the prover's completeness. So we simply delay the instantiation phase. This modification substantially reduces the number of replacement instances and the number of relevant literals created, and results in marked performance improvement. Further, in our experience there is not a significant drop in the number of theorems proved.

In the ground replacement version of RRTP is shown in Figure 3.3. There are

two phases: *replacement* and *instantiation*. At the beginning of the replacement phase, as before, replacement rules are created from the input clauses. These rules are then used to generate replacement instances. Some of the instances may already be ground and are tested for unsatisfiability. After a few rounds of replacement, the prover times out and enters the instantiation phase. During the instantiation phase, the non-ground replacement instances are systematically grounded and tested for unsatisfiability.

Size Increasing Replacement Instances

Consider the replacement rule

$p(X) \rightarrow p(f(X)), p(g(X))$

Any replacement instance from this rule is size-increasing as a relevant literal created using this replacement rule is larger than the antecedent literal. Further, suppose that $p(a)$ is some relevant literal, then we get the following relevant literals:

$p(a), p(f(a)), p(g(a)), p(f(g(a))), p(g(f(a))), \ldots, p(f(\ldots))$

in successive rounds of replacement by the same replacement rule. Typically, such literals create larger and larger instances that do not help the proof.

Definition 3.3.3 *A replacement instance is said to be size-increasing if some consequent literal is larger than all of the antecedent literals. The consequent literals of size-increasing replacement instances are termed as size-increasing literals.*

To control the effect of such literals on the proof search, the addition of size-increasing literals to the set of relevant literals is delayed until no more replacements can be performed. This restriction, usually allows the prover to go more rounds, and find deep proofs that do not involve size increasing relevant literals. Of course, proofs involving size increasing relevant literals are delayed.

Equality and Brand's Transformation

One way for a theorem prover to handle equality is to simply use the equality axioms presented in Chapter 2, definitions 2.2.29,2.2.30. However, this approach is not very suited to the way RRTP functions. The chief problem is that the substitution axioms for the function and predicate symbols tend to outnumber axioms that are more central to the theorem being proved. As a result, several relevant literals that contribute little to the

3.4 Performance

3.4.1 TPTP Library

Geoff Suttcliffe and Christian Suttner have in [SS97] described a classification of theorems under four categories: easy— solved by all state-of-the-art ATP systems; difficult— solvable by some state-of-the-art ATP systems; unsolved— solvable by no state-of-the-art ATP system and open— it is not known whether the problem is a theorem. Of the thousands of problems archived in [SSY93], around five hundred are categorized as difficult. These problems exemplify what is currently within the reach of ATP technology. In most cases, there are only one or two provers that can prove these selections.

The problems are categorized under several domains. For a detailed introduction to these domains and the problems we refer the reader to [SSY93].

We summarize results of the performance of RRTP on these problems in Table 3.1, in comparison with some other provers. Since the provers did not all run on the same architectures, we do not include the times each problem took. We only present summary information in this table. A more detailed table is presented in Appendix A. Expectedly, RRTP does very well in theorems involving set theory (SET). Replacement is a natural way to solve problems in the von Neumann-Bernays-Godel set theory where many predicates are defined in terms of other predicates. The performance of RRTP on problems not involving equality is even more encouraging. This is shown in Table 3.2. [1]

We briefly comment on the provers compared. Otter[McC90] is the state-of-the-art resolution prover. This theorem prover is the culmination of over thirty years of ATP research and development in Argonne National Labs. Recently, Argonne National Labs enjoyed some some publicity[Kol96] as one of their provers proved an open conjecture in mathematics— the Robbins problem. SETHEO[LSBB92] is a theorem prover based on the idea of model elimination[Lov78]. The proof procedure is implemented as an extension of the Warren Abstract Machine. It is a fairly substantial software product that incorporates many ideas in theorem proving research. The pervading philosophy in SETHEO is that "one ingenious idea is not sufficient; the engineering aspect forms a substantial component of the system". As a result it is difficult to pinpoint the reason for SETHEO's performance. A

[1] From the theorems used in the CADE-14 Automated Theorem Proving Competition, when only non-Horn non-Equality problems are considered, RRTP exhibits performance that was superior to all the other systems.

Comparison of Some Provers on difficult TPTP problems							
Domain	# Problems	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
BOO	13	11	5	4	1	9	6
CAT	32	26	12	10	10	22	17
CID	2	2	0	0	0	0	0
COL	6	1	2	1	1	2	1
GEO	31	9	5	5	11	7	9
GRP	35	16	8	6	7	10	3
HEN	20	20	2	4	7	9	5
LAT	2	2	0	0	0	0	0
LCL	23	18	21	13	3	19	16
LDA	1	1	0	1	1	1	0
MSC	1	1	1	0	1	1	1
NUM	25	11	10	7	2	9	11
PLA	23	0	23	0	2	0	0
RNG	13	13	4	5	4	8	5
ROB	2	2	0	0	0	1	0
SET	116	55	33	11	0	53	62
SYN	37	35	26	35	35	35	35
Total	382	223	152	102	85	186	171

Table 3.1: Comparison of RRTP with other provers

version of SETHEO incorporating equality won the CADE-13 Automated Theorem Proving System Competition. Linus[Let97],CLIN and CLIN-E are all theorem provers based on hyper-linking. CLIN-E uses a smallest instance preference strategy. Linus, like RRTP, incorporates UR-Resolution as well.

3.5 Conclusions

RRTP does extremely well on near-propositional problems, and range-restricted problems. It also exhibits superior performance on non-Horn non-Equality problems. The prover is ineffective on problems where the theorem is a non-ground clause. This is especially apparent with the planning problems (PLA) from the TPTP. We address this class of problems with a proof procedure for Horn theories described in the next Chapter.

Furthermore, replacement can be used with any theorem prover; given any theorem, any prover may create some replacement instances from the input clauses and then proceed with its own proof procedure. We have observed that this improves performance

Difficult TPTP problems without equality						
Domain	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
GRP (1)	1	0	0	1	1	1
LCL (21)	17	**20**	12	3	18	16
NUM (6)	4	5	5	1	4	6
PLA (22)	0	**22**	0	1	0	0
SET (13)	4	7	1	0	6	**13**
SYN (35)	35	26	35	35	35	35
Total (98)	61	**80**	53	41	64	71

Table 3.2: Comparison with other provers on problems not involving equality

in provers like CLIN-S[Chu94b].

Chapter 4

A Proof Procedure for Horn Theories

4.1 Introduction

There is considerable interest in decision and semi-decision procedures for sublogics of first-order logic. Of these sublogics, Horn theories are particularly interesting because Horn clause logic can be viewed as a programming language, and Horn theories resemble programs. Horn clauses are first-order clauses that have only one positive literal, and therefore can be viewed as implication rules. A clause of the form, $L_1, \ldots, L_n \rightarrow R$, is interpreted as the definition of the procedure R in terms of the subprocedures $L_1, \ldots, L_n$. This interpretation forms the basis of logic programming and is realised in the language Prolog. Several applications are rooted in Horn clause logic. The most notable among these is logic programming, with a well-understood theory[Apt90] and the programming language Prolog [CM81].

The attraction of logic programming is that it is declarative— the program is simply a set of rules or clauses. As far as the programmer is concerned, ideally, the execution of such programs is left to methods that maintain the declarative meaning. Changing the order of the clauses or the order of the literals within the body of the clause does not change the declarative meaning should not cause the execution methods to behave differently. Such a system allows the programmer to convey specifications simply that can be directly executed.

Prolog was invented to serve this declarative need. Although Prolog is an admirable declarative language, it deviates from the ideal in many ways. The most notable deviation is that Prolog does not have declarative control. This implies that the meaning of a Prolog program changes with the order in which the clauses are provided in the program. Reordering the clauses in one program may change it to another. Other problems such as the use of negation as failure, and unification without occurs-check make Prolog unusable for classical Horn logic[Pla84].

Since Horn clause logic is only partially decidable, there has been interest in sublogics of Horn clause logic. Several restricted forms of Horn clause logic form the basis for many decidable, tractable, database languages. Horn clause logic has also been used to encode plans as they provide a natural means for expressing rules of cause and effect. There has been considerable work in the database community [CGT90] towards languages that have simple, efficient and terminating approaches. Such languages are based on restricted Horn theories. In fact, the technique we present resembles Backchain Iteration[Wal93], which is a decision procedure for stratified datalog programs. Datalog programs are logic programs that have only variables and constants.

Theorem proving techniques such as binary resolution do not distinguish the goal and usually end up creating all consequences of the input. This is pathological for problems such as planning where the objective is to come up with a plan given certain plan criteria in terms of some clauses, and the input configuration as unit clauses. A purely forward chaining strategy such as positive hyper-resolution[McC90] results in all the possibilities that may be derivable from the initial configuration. It is often the case that there are too many states that are reachable from the initial configuration, and this approach is unsuited for determining a plan for a specific action. A purely backward chaining strategy such as negative hyper-resolution results in clause combination and redundant searches. It seems to be the case that most strategies are either inefficient on Horn clauses or are not sensitive to the theorem being proved. It seems unlikely that simply combining general axioms will make much progress to proving the theorem[Pla94]. On the other hand, working backwards from the goal using all-negative resolution seems to be highly inefficient for Horn clauses.

The motivation for our work continues from previous efforts for such a decision procedure [Pla82],[Pla88a]. We are interested in a sound and complete, goal sensitive, proof procedure that does well for first-order Horn theories. Our strategy can broadly be classified as combining forward and backward chaining, but the novel aspect is that we interleave the

two strategies so that we work from the goal to produce instances of the input clauses, and use forward reasoning within these instances for proving the theorem. It is instance-based and refutational in nature. Since Horn theories are in general only partially decidable, this strategy may not terminate if there is no proof. However it is possible to bound the search by some other measure, for example when using it to generate a plan, information about the length of the plan may be used to bound the search.

This chapter is organized as follows: we first go over the terminology used in the rest of the chapter, following which we describe our algorithm in detail. After proving the soundness and completeness of the basic algorithm, we present some refinements that make it more efficient from a practical perspective. We present results of running this method and compare it with the results of some other theorem provers.

4.2 Background and Definitions

We briefly overview the terminology used in the rest of the chapter. A *term* is a well-formed expression composed of variables and function and constant symbols. For example, $f(x, g(a, b))$ is a term. An *atom* is a predicate symbol, that is either a propositional constant, or has terms for arguments. For example, P and $Q(f(x, g(a, b), z))$ are atoms. A *literal* is an atom or an atom preceded by a negation sign. P and $\neg Q(f(x, g(a, b), z))$ are literals. A *positive* literal is one that has no negation sign preceding it, and a *negative* literal is one that has a negation sign preceding it. A *clause* is a disjunction of literals, usually written as a set for brevity. We consider only *skolemized* clauses, that is the variables in the clauses are all implicitly universally quantified. Since the skolemization process preserves unsatisfiability and our proof-procedure is refutational, skolemizing the clauses is acceptable to us. For example the formula $(\forall x)(\forall y)((\neg Q(f(x, g(a, b), z)), P(x)) \wedge (R(x, f(x)), \neg S(y)))$ is skolemized to $\{\{\neg Q(f(x, g(a, b), z)), P(x)\}, \{R(x, f(x)), \neg S(y)\}\}$. The symbols a and b are known to be constants. Variable names in clauses are freely renamed to avoid confusion.

A *substitution* is a mapping from variables to terms. When applied to a clause, a substitution replaces the variables of the literals of the clause with the corresponding terms. The result is an *instance* of the clause. For example, when the substitution $\{x \mapsto f(a), y \mapsto x\}$, is applied to the clause $\{P(x, y), \neg Q(y, y)\}$, we get the clause instance, $\{P(f(a), x), \neg Q(x, x)\}$. A literal A is said to be *more general* than another literal B if there exists a substitution that when applied to A results in B. Two literals are *unifiable* if there

exists a substitution that when applied to the literals creates the same literal instance. Such a substitution is called *unifier* of the literals. The *most general unifier* of two literals is at least as general as any other literal. The definitions for *general*, *unifiable* and *most general unifier* are extended to clauses as well.

A clause is *all-negative* if it has only negative literals. A clause is *all-positive* if it has only positive literals. A *unit clause* has only one literal. A *Horn clause* is a clause that has at most one positive literal. We represent Horn clauses that have at least one positive literal and are not unit clauses, such as $\{\neg L_1, \ldots, \neg L_k, R\}$, where $L_1, \ldots, L_k, R$ are positive literals, as $L_1, \ldots, L_k \rightarrow R$. The natural reading of this representation is that R is a logical consequence of $L_1 \wedge \ldots \wedge L_k$. We refer to $L_1 \wedge \ldots \wedge L_k$ as the *antecedent* literals, and R as the *consequent* literal. An instance of a Horn clause is a *Horn instance.* A *Horn theory* is a theory that is made of ony Horn clauses. A *logic program* is a Horn theory that has no all-negative clauses. A *goal* is an all-negative clause. It is usually a unit clause.

A logic program P entails a goal G if and only if G is a logical consequence of P, or if $P \wedge \{\neg G\}$ is unsatisfiable. A strategy that proves theorems by contradiction is termed *refutational.* A clause $C \in P$ is said to be *relevant* to a proof if $P \wedge \{\neg G\}$ is unsatisfiable, but $\{P \setminus C\} \wedge \{\neg G\}$ is satisfiable.

Given a set of Horn clauses, we define the *backward depth* of a clause instance C inductively: If C is an all-negative clause then the backward depth of C is zero. Suppose that C is a Horn clause of the form $L_1, \ldots, L_k \rightarrow R$. If R unifies with the negative literal of a Horn instance that has a backward depth of i, by a unifier Θ, then $C\Theta$ has a backward depth of $i+1$. Note that the backward depth of a clause instance is not necessarily unique. *Forward Depth* of a positive unit is defined as follows: If L_i is a positive unit clause, then forward depth of L_i is zero. Suppose $C\Theta$ is an instance of the input clause C of the form $L_1, \ldots, L_k \rightarrow R$. Suppose further that $M_1, \ldots, M_k$ unify with $L_1, \ldots, L_k$ by the unifier Θ, then the forward depth of $R\Theta$ is $1 + max(fd(M_1), \ldots, fd(M_k))$, and is given by $fd(R\Theta)$. where $fd(M_i)$ is the forward depth of literal M_i. The forward depth of a unit is not unique either.

The following example illustrates the above definitions.

Example. Consider the following set of Horn clauses:

$\{\neg P(f(x))\}$
$\{S(g(y)), T(y) \rightarrow P(y)\}$
$\{S1(x), S2(x) \rightarrow S(x)\}$
$\{T1(x), T2(x) \rightarrow T(f(x))\}$
$\{T1(a)\}$
$\{T2(a)\}$
$\{T(f(a))\}$

The clause instance $\{\neg P(f(x))\}$ has a backward depth of zero. As $P(f(x))$ and $P(y)$ are unifiable by the substitution $\{y \mapsto f(x), x \mapsto x\}$, the instance $\{S(g(f(x))), T(f(x)) \rightarrow P(f(x))\}$ has a backward depth of one. Similarly, $\{S1(g(f(x))), S2(g(f(x))) \rightarrow S(g(f(x)))\}$ has a backward depth of two. The literals $T1(a)$ and $T2(a)$ have forward depth zero, and the literal $T(f(a))$ has forward depth both one and zero.

4.3 An Instance-based Proof Procedure

In this section, we describe an instance-based proof procedure for Horn clauses. We refer to this procedure as *Horn-prover*. We assume that there is only one all-negative clause; moreover, this clause has only one literal. This allows us to view the set of Horn clauses as a logic program P and a goal G. Goal sensitive proof procedures for logic programs work backward from the goal by constructing subgoals. In SLD-resolution, this is done by resolving the goal clauses with clauses in the program. Implementation of SLD-resolution is convenient if a depth first approach is used, and this leads to incompleteness as well.

Our proof procedure is goal-sensitive as well. Instead of creating subgoals, however, the *Horn-prover* creates instances of the input clauses, of increasing backward depth starting from the goal. The process of creating the instances resembles backward chaining; we refer to it as *backchain*. The *Horn-prover* periodically searches among these instances for the proof. This search resembles forward chaining proof procedures such as positive hyper-resolution, or positive unit resolution for Horn clauses. We refer to this as *forwardChain*. We overview how *backchain* and *forwardchain* are interleaved: After every round of *backchain* , upon which several instances of some backward depth are created, a round of *forwardchain* is performed. Each round of *forwardchain* is made of several levels of a controlled form of unit resolution. The number of such levels is the same as the number of rounds of *backchain* performed thus far.

4.3.1 BackChain

We define the procedure *backchain*. *backchain* takes as input a set of Horn clauses, $\mathcal{H}$, and a set of Horn instances, $\mathcal{I}$. For every clause $L_1, \ldots, L_k \rightarrow R \in \mathcal{H}$, suppose there exists an instance in $\mathcal{I}$ that contains R' as an antecedent literal, and R and R' unify with most general unifier Θ, then *backchain* outputs the clause instance $\neg L_1\theta, \ldots, \neg L_k\theta \rightarrow R\theta$.

If $\mathcal{I}$ is empty then *backchain* trivially returns all the all-negative clauses in $\mathcal{H}$. The procedure is called as *backchain*($\mathcal{H}$, $\mathcal{I}$). We use *backchain* as follows: In each round, the input clauses, including the goal, and instances of backward depth k are used as input to *backchain*. This produces, by definition, instances of backward depth $k+1$. During the first round, *backchain* picks the goal clause from the input set.

We illustrate *backchain* with the following example. Consider the following clauses belonging to a set of Horn clauses.

$\{p1(X,Y,Z), \neg q1(X,Y), \neg q2(X,Z)\}$
$\{p2(X,Y,Z), \neg q1(X,Y), \neg q3(X,Y)\}$
$\{p3(X,Y,X), \neg q2(X,Y), \neg q3(X,Y)\}$

Let the following clause instances be instances with backward depth k

$\{r1(X,Y), \neg p1(X, f(X), Y)\}$
$\{r2(X,Y), \neg p1(X,Y, f(Y))\}$
$\{r3(X,a), \neg p3(X,X,a)\}$

From the above two as input, *backchain* creates the following instances. Their backward depth is $k+1$.

$\{p1(X, f(X), Y), \neg q1(X, f(X)), \neg q2(X,Y)\}$
$\{p1(X,Y, f(Y)), \neg q1(X,Y), \neg q2(X, f(Y))\}$
$\{p3(a,a,a), \neg q2(a,a), \neg q3(a,a)\}$

4.3.2 ForwardChain

We now describe *forwardchain*. It is very similar to positive unit resolution, however the procedure is divided into several levels. We describe a single level of *forwardchain*.

forwardchain takes as input a set of Horn instances, $\mathcal{I}$, and a set of positive unit literals, $\mathcal{P}$. It is invoked as *forwardchain* ($\mathcal{I}$, $\mathcal{P}$). Each level of *forwardchain* proceeds as

follows: For each clause instance C, of the form $L_1, \ldots, L_k \rightarrow R$, in $\mathcal{I}$, suppose there exist positive literals $P_1, \ldots, P_k$ in P, such that $L_1, \ldots, L_k$ and $P_1, \ldots, P_k$ unify with most general unifier θ then, $R\theta$ is output or that $R\theta$ is proved. If there are unit positive clauses in $\mathcal{I}$, then *forwardchain* naturally outputs them.

4.3.3 The Prover

We now describe the algorithm *Horn-prover*. It is basically a combination of *forwardchain* and *backchain* : Each round of *Horn-prover* is made of one round of *backchain* and one round of *forwardchain*. In the dth round of *Horn-prover* , *backchain* creates instances of backward depth $d-1$. The first level of *forwardchain* takes as input the instances of the greatest backward depth —in this case $d-1$, and no positive units. The resulting positive units, and instances having backward depth $d-2$, are the input to the next level of *forwardchain*. The total number of levels in *forwardchain* is the same as the number of rounds of *Horn-prover* taken place so far, this being d in this case. If the final level of *forwardchain* outputs $\perp$, then we have a proof in the dth round of the *Horn-prover*. Otherwise, *Horn-prover* moves on to the next round. The algorithm is shown is Figure 4.1. We illustrate *Horn-prover* using the following example.

Example. Consider the following set of Horn clauses.

$\{\neg P(f(x))\}$
$\{P1(x), P2(x) \rightarrow P(x)\}$
$\{Q1(x), Q2(x) \rightarrow P(x)\}$
$\{P3(x), P4(x) \rightarrow P1(x)\}$
$\{Q3(x), Q4(x) \rightarrow Q1(x)\}$
$\{P2(f(a))\}$
$\{P3(f(a))\}$
$\{P4(x)\}$
$\{Q2(f(g(a)))\}$

The first round of *backchain* produces S_0, instances of backward depth zero:

$\{\neg P(f(x))\}$

forwardchain outputs nothing at the lowest level.

Algorithm 3 (HornProver, Basic Description)
Input: Set S of Horn Clauses.
Output: Proof of Unsatisfiability of S.

```
1   S_0 ← backchain(S, φ);
2   k ← 0;
3
4   while not PROOF do
5      k ← k + 1;
6      S_k ← backchain(S, S_{k-1});
7      F_k ← φ;
8      for i = k  to i = 0 do
9         F_{i-1} ← forwardChain(S_i, F_i);
10     enddo
11     if ⊥∈ F_0 then
12        PROOF ← true;
13     endif
14  endwhile
```

Figure 4.1: A proof procedure for Horn clauses

The next round of *backchain* produces S_1, the instances of backward depth one:

$\{P1(f(x)), P2(f(x)) \rightarrow P(f(x))\}$
$\{Q1(f(x)), Q2(f(x)) \rightarrow P(f(x))\}$

Again, *forwardchain* outputs nothing at its first level. Due to there being no units created at this level, the activity of *forwardchain* at the next level with backward instances of depth zero, will be no different from the previous round. Therefore, there is no need to continue to the next level.

The third round of *backchain* produces S_2, the following instances having backward depth two:

$\{P3(f(x)), P4(f(x)) \rightarrow P1(f(x))\}$
$\{P2(f(a))\}$
$\{Q2(f(g(a)))\}$

forwardchain outputs F_2, the positive units $Q2(f(g(a)))$ and $P2(f(a))$ at this level. With these units and the instances of backward depth one, the next level of *forwardchain* outputs nothing.

Next, *backchain* produces S_3— instances having backward depth three:

$\{P3(f(a))\}$
$\{P4(f(a))\}$

forwardchain outputs the following positive units from S_3: $P3(f(a))$ and $P4(f(a))$. Using these units and S_2, *forwardchain* outputs $P1(f(a))$, $P2(f(a))$, and $Q2(f(g(a)))$ at the next level. This produces $P(f(a))$, which finally at the last level with the single instance from S_0, *forwardchain* outputs, F_0 which includes $\perp$.

Note that *forwardchain* differs from unit resolution in a couple of aspects. Firstly, as seen from the example, positive unit literals do not indiscriminately resolve with the backward instances. More specifically, positive units created at level F_i combine only with backward instances S_i. Secondly, the resolution is done on an instance only if all the negative literals of the instance can be resolved away. In this respect *forwardchain* is more like

positive hyper-resolution on Horn clauses. Furthermore, note that in every round *forward-chain* repeats the work done on previous rounds. We address this and other improvements later.

4.4 Completeness and Soundness

In this section, we show that *Horn-prover* is sound and complete. The soundness of *Horn-prover* is trivially established by showing that it is a restricted form of unit resolution.

Theorem 4.4.1 Horn-prover *is sound.*

Proof. The proof of unsatisfiability is found during *forwardchain* . Recall that *forward-chain* is a restricted form of unit resolution. In *forwardchain* unification takes place only between certain positive units and negative literals of the input clause instances. All of these unifications can be emulated in unit resolution. If *forwardchain* outputs $\perp$, then unit resolution derives the empty clause. Since unit resolution is sound [CL73], *Horn-prover* is sound. □.

Showing completeness of *Horn-prover* is more complicated. We develop it by proving some lemmas first. From Herbrand's theorem, if a set of first-order clauses is unsatisfiable, then there exists a finite set of ground instances of these clauses that are unsatisfiable. Furthermore, from a set of unsatisfiable ground Horn clauses, G_H, it is possible to obtain the following semantic tree T: The nodes of T are all unsigned literals, or two special symbols, $\top$ ("true") and $\perp$ ("false"). For each non-leaf node N, and its children $N_1, \ldots, N_k$, there exists an instance $N_1, \ldots, N_k \rightarrow N$, in G_H. The root of the tree is $\perp$ and every leaf is $\top$. We refer to an instance to be at depth d in T when the consequent literal in the instance is at depth d. The following example illustrates the idea of the semantic tree.

Example. Consider the following set of ground unsatisfiable clauses.

$\{\neg P\}$ (or $\{P \rightarrow \perp\}$)
$\{P1, P2, P3 \rightarrow P\}$
$\{P4, P5 \rightarrow P1\}$
$\{P1, P5 \rightarrow P3\}$
$\{P2\}$ (or $\{\top \rightarrow P2\}$)
$\{P4\}$

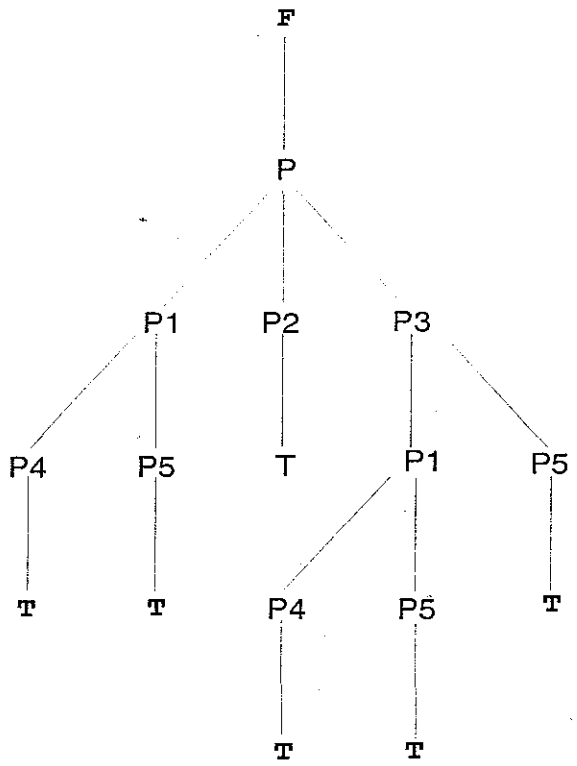

Figure 4.2: An example proof tree

$\{P5\}$

The semantic tree that corresponds to these clauses is shown in Figure 4.2. □

We use the idea of this semantic tree to prove completeness of *Horn-prover* . We refer to the unsatisfiable set of Horn clauses as S. The semantic tree made of ground instances of S is denoted by T. The proof is divided into two lemmas. We first show that *backchain* selects instances of S that are either in T, or are more general than the instances in T. We then show that *forwardchain* creates the empty consequent literal ($\perp$) from the instances selected by *backchain* .

Lemma 4.4.2 *For every ground instance, I_G at depth d in T, there exists an instance of S, I_B, that has a backward depth d, and is more general than I_G,*

Proof. We prove this by induction on d. The basis is as follows. The root of T is $\perp$. Therefore, the instance at depth zero is an instance of some all-negative clause in S. From the definition of *backchain* , all-negative clauses in S have backward depth zero. The basis

for induction holds.

The induction hypothesis is that, for every ground instance I_G in T at depth d, there exists an instance, I_B of S, with backward depth d, such that I_B is more general than I_G. We now show that for every ground instance at depth $d+1$ in T, there exists an instance of S with backward depth $d+1$.

Suppose that there exists an instance, $G_1, \ldots, G_k \rightarrow G$, at depth $d+1$ in T. Since $G_1, \ldots, G_k \rightarrow G$ is an instance of some clause in S, there exists a clause in S $L_1, \ldots, L_k \rightarrow L$, that is the same or more general than $G_1, \ldots, G_k \rightarrow G$. Trivially, G unifies with the literal L.

The parent of G in T is some node H at depth d. The ground instance I_H at this node is of the form $\ldots, G, \ldots \rightarrow H$. By the induction hypothesis, there exists an instance of backward depth d, that is more general than I_H. Therefore, it contains an antecedent literal B that is more general than G. Since B is more general than G, B must also unify with L, with most general unifier, say, Θ. The clause instance, $L_1, \ldots, L_k \rightarrow L\Theta$, therefore has backward depth $d+1$. □

We refer to the instances created by *backchain* that are more general than the ground instances in the T as *relevant* instances. We next show that from the relevant instances *forwardchain* creates $\perp$. We first show that *forwardchain* creates $\perp$in the ground case and then lift the idea.

Lemma 4.4.3 *Given a set of instances, containing more general instances than every instance in T,* forwardchain *outputs $\perp$ after d levels, where d is the height of T.*

Proof. First, we show that given the ground instances in T, with their depth in T denoting their backward depth, *forwardchain* produces $\perp$after d levels. It follows from the definition of *forwardchain* that given an instance $G_1, \ldots, G_k \rightarrow G$, and the positive literals $G_1, \ldots, G_k$, *forwardchain* outputs G. From this, and the definition of the semantic tree, *forwardchain* outputs the literals at depth k in T, with literals at depth k and instances in depth $k-1$ as input. Therefore, starting with the instances at depth $d-1$, which are all positive units, *forwardchain* outputs the nodes of T from the bottom to the empty consequent literal at the top of the tree.

Now we extend the above results to non-ground relevant instances. If *forwardchain* produces G from $G1, \ldots, Gk \rightarrow G$, and positive literals $G1, \ldots, Gk$, then, from a more

general instance, $B1, \ldots, Bk \rightarrow B$, and positive literals, $S1, \ldots, Sk$, *forwardchain* produces a literal that is more general than G. Any literal more general than $\perp$ is still $\perp$. □

Theorem 4.4.4 Horn-prover *is complete.*

Proof. This follows from Lemma 4.4.2 and Lemma 4.4.3. □

4.5 Refinements

4.5.1 Caching Forward Units From the Input Clauses

Typically, the number of backward instances created each round grows substantially. That is, as k increases the number of instances in S_k increases, sometimes to the extent that computing them is a long task. We can precompute some of the consequences of the input positive units and cache them. That is, we compute all positive units up to a predetermined forward depth. This is usually quick and does not increase the search space, and trims the number of levels of the proof. Once precomputed, the positive units are assigned forward depths of zero.

4.5.2 Caching Forward Lemmas and Deleting Duplicate Instances

One of the problems of the procedure is that, every round, all of the work done in the previous rounds of *forwardchain* is repeated. For example, consider again the proof in Figure 4.3. The proof takes four rounds. The literal $P1$, marked with a square, that appears in the second level, is proved in the third and the fourth round of *Horn-prover*.

In most theorem proving procedures, the work done in each round is substantially more than the work done in all of the previous rounds, and effort to curb the repetition is usually not worth the payoff. However, in this case, it is worthwhile to prove every forward unit only once and cache that unit as a consequence of the input. Note also that another occurrence of the literal $P1$, circled in the Figure 4.3, causes the same proof to be repeated elsewhere. We would like to avoid having to repeat proofs of the identical literals regardless of where they appear.

We first discuss how to avoid repeating proofs of the same literal during different rounds of the algorithm. Recall that *forwardchain* takes as input a set of Horn instances, and a set of positive literals. We modify the input to be a set of Horn instances, $\mathcal{I}$, and two

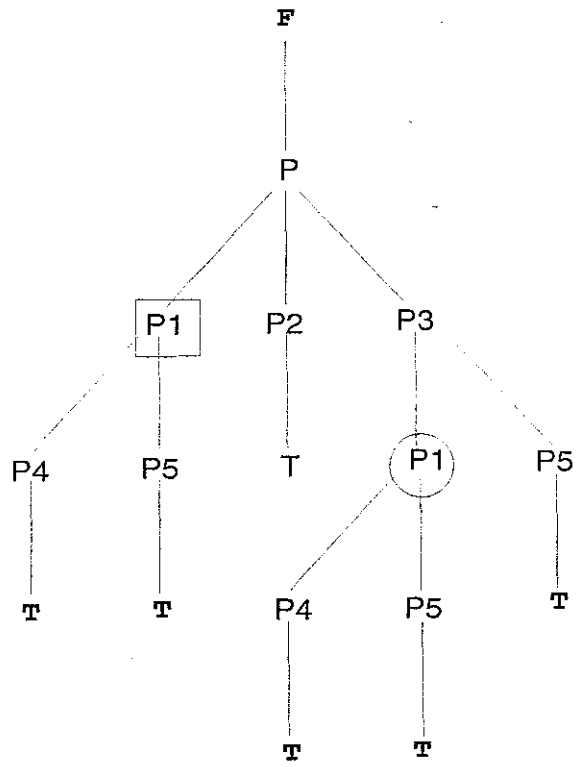

Figure 4.3: An Example Proof

Algorithm 4 (HornProver, Caching Forward Units)
Input: Set S of Horn Clauses
Output: Proof of Unsatisfiability of S

```
1   S_0 ← backchain(S, φ);
2   k ← 0;
3
4   while not PROOF do
5      k ← k + 1;
6      S_k ← backchain(S, S_{k-1});
7      F_k ← φ;
8      C_k ← φ;
9      for i = k  to i = 0 do
10        C_{i-1} ← forwardChain(S_i, C_i, F_i);
11        F_{i-1} ← C_i ∪ F_i;
12     enddo
13     if ⊥∈ F_0 then
14        PROOF ← true;
15     endif
16  endwhile
```

Figure 4.4: Caching positive units to avoid recreating them every round

sets of positive literals, $\mathcal{P}_1$ and $\mathcal{P}_2$. It is invoked as *forwardchain* $(\mathcal{I}, \mathcal{P}_1, \mathcal{P}_2)$. Each level of *forwardchain* now proceeds as follows:

For each clause instance $C \in \mathcal{I}$, of the form $L_1, \ldots, L_k \rightarrow R$, suppose there exist positive literals $P_1, \ldots, P_k$ in $\mathcal{P}_1 \cup \mathcal{P}_2$, such that $L_1, \ldots, L_k$ and $P_1, \ldots, P_k$ unify with most general unifier Θ, and that at least one of $P_1, \ldots, P_k$ is taken from $\mathcal{P}_1$, then $R\theta$ is output at this level. *forwardchain* returns the positive units in $\mathcal{I}$ only if $\mathcal{P}_1$ and $\mathcal{P}_2$ are both empty.

The new algorithm is shown in Figure 4.4 As in the earlier version, a positive unit created from an instance of backward depth i belongs in F_i. At every level, forward units created at this round are separated from the ones created in earlier rounds using C_i and F_i. The new definition of *forwardchain* , which now takes two sets of positive units, ensures that if a positive unit is output at level j, and its forward depth is d, then it is output in the $j + d$th round of *Horn-prover*.

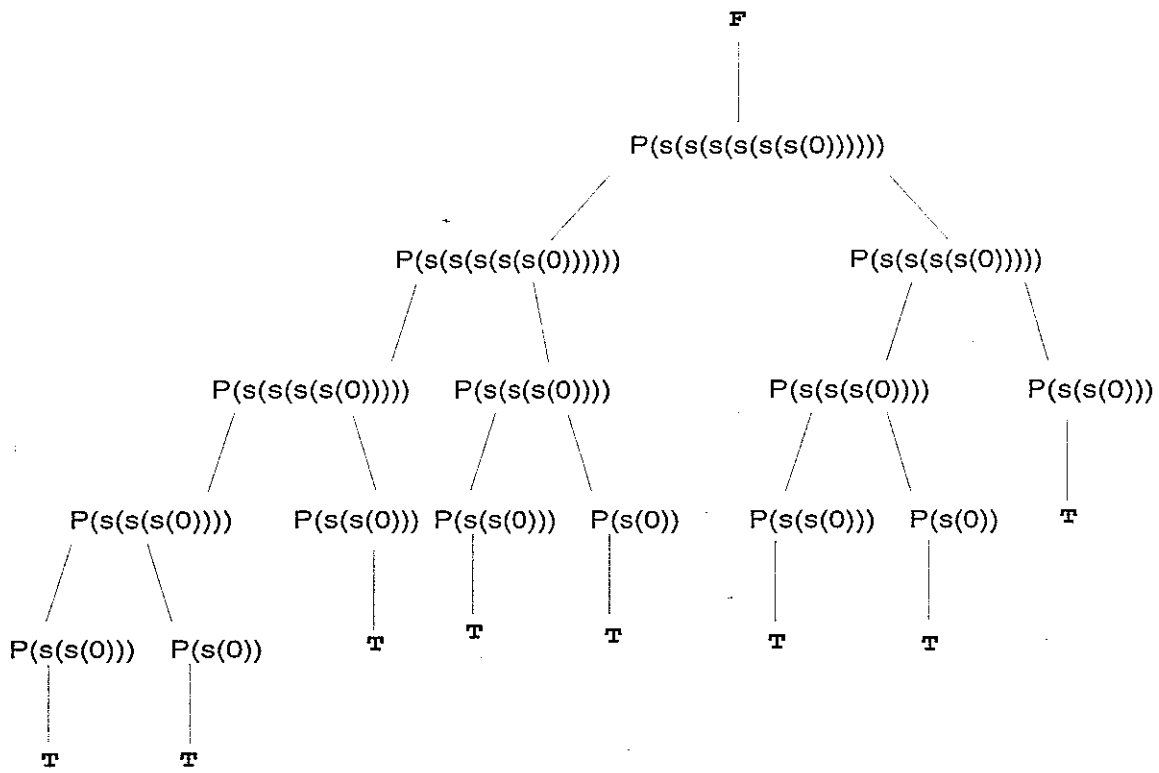

Figure 4.5: Fibonacci Example

Now we direct our attention to avoiding repeating proofs for the same literal although they may appear at different positions in the tree. The need is best illustrated by the following example modeled after generating Fibonacci numbers.

Example. Consider the following unsatisfiable Horn clauses:

$\{P(s(s(X))), \neg P(s(X)), \neg P(X)\}$
$\{P(s(0))\}$
$\{P(0)\}$
$\{\neg P(s(s(s(s(s(0))))))\}$

□

The semantic tree corresponding to the proof of unsatisfiability is shown in Figure 4.5.

Consider the instances,
$\{P(s(s(s(s(0))))), P(s(s(s(0)))) \rightarrow P(s(s(s(s(0)))))\}$ and

$\{P(s(s(0))), P(s(s(s(0)))) \rightarrow P(s(s(s(s(0)))))\}$
at level two. The antecedent literals, $P(s(s(s(0))))$, marked in the figure, appear in both these instances, and appear at the same depth in the tree. Clearly, it is not necessary to have two copies of the backward instance $\{P(s(0)), P(s(s(0)) \rightarrow P(s(s(s(0)))))\}$, at this backward depth; to prove these units during *forwardchain* , backward instances created from only copy is needed, and the other can be deleted. This idea can be extended to keeping only the most general instance at any level.

Now consider the positive unit $P(s(s(s(0))))$ in the lower level as shown in the figure. Since it is in a different level than the other two occurrences of $P(s(s(s(0))))$, for the instance $\{P(s(0)), P(s(s(0)) \rightarrow P(s(s(s(0)))))\}$ to be deleted, some bookkeeping needs to be done.

An instance is deleted provided a more general instance of lesser backward depth is already present. The record of deleted instances is kept associated with the more general instance and the backward depth of the more general instance. Whenever *forwardchain* creates a positive unit from the most general instance, the unit is made available at the appropriate levels for use in the next round of the *Horn-prover*.

4.6 Experimental Results

There are about thirty planning problems from the TPTP[SSY93] collection. We ran *Horn-prover* on them and have tabulated the results of the run, and have compared it with other provers as indicated in Table 4.1. Otter [McC90] is a resolution-based prover that uses forward chaining for Horn problems. It is interesting to see that it gets very few problems; the long standing belief is that resolution-based forward chaining methods are very well suited to Horn clauses. We believe the lack of goal-sensitivity to be the chief reason for Otter's poor performance. SETHEO [LSBB92] is a sophisticated prover involving many strategies but the basic idea is model elimination. Linus[Let97] and CLIN-S[Chu94a] are both clause linking provers. Since the provers ran on different architectures we do not present the times taken, however it is worth mentioning that all the provers, except CLIN-S, take only a few seconds to get the proofs. CLIN-S takes several minutes to get the proofs.

The proof procedure for Horn theories is incorporated in RRTP. The Horn problems are addressed by this technique. Table 4.2 reflects the superior performance of RRTP when combined with the Horn procedure on eligible problems.

Comparison of Planning Problems over several Provers					
Problem	BackChain	Otter	SETHEO	LINUS	CLIN-S
PLA001-1	Y	N	Y	Y	Y
PLA002-1	Y	Y	Y	Y	Y
PLA002-2	N	N	N	N	Y
PLA003-1	Y	Y	Y	Y	Y
PLA004-1	Y	N	Y	N	Y
PLA004-2	Y	N	Y	N	Y
PLA005-1	Y	N	Y	N	Y
PLA005-2	Y	N	Y	N	Y
PLA006-1	Y	Y	Y	Y	Y
PLA007-1	Y	N	Y	N	Y
PLA008-1	N	N	Y	N	N
PLA009-1	Y	N	Y	N	Y
PLA009-2	Y	N	Y	N	Y
PLA010-1	N	N	Y	N	N
PLA011-1	Y	N	Y	N	Y
PLA011-2	Y	N	Y	N	Y
PLA012-1	N	N	Y	N	N
PLA013-1	Y	N	Y	N	Y
PLA014-1	Y	N	Y	N	Y
PLA014-2	Y	N	Y	N	Y
PLA015-1	N	N	Y	N	N
PLA016-1	Y	N	Y	N	Y
PLA017-1	Y	Y	Y	Y	Y
PLA018-1	N	N	Y	N	N
PLA019-1	Y	N	Y	N	Y
PLA020-1	Y	Y	Y	Y	Y
PLA021-1	Y	N	Y	N	Y
PLA022-1	Y	N	Y	N	Y
PLA022-2	Y	N	Y	N	Y
PLA023-1	N	N	Y	N	N
Total	23	5	29	6	24

Table 4.1: A comparison of different provers on Horn problems

Difficult TPTP problems without equality						
Domain	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP(Horn)
GRP (1)	1	0	0	1	1	1
LCL (21)	17	**20**	12	3	18	16
NUM (6)	4	5	5	1	4	6
PLA (22)	0	**22**	0	1	0	17
SET (13)	4	7	1	0	6	**13**
SYN (35)	35	26	35	35	35	35
Total (98)	61	80	53	41	64	**88**

Table 4.2: A comparison of different provers on non-Equality problems

4.7 Extensions and Conclusions

The contribution of this chapter is a proof procedure for Horn theories that combines backward chaining with forward chaining techniques. Our view is that good theorem proving strategies for Horn clauses should be goal-sensitive without having the usual inefficiencies associated with backward chaining strategies. We believe that the procedure we present illustrates this view. We have not analytically established the efficiency of this method to be superior to non-goal sensitive techniques, but we believe that the performance comparison on Horn problems to be some indication of our claim. We also have some ideas on extending this to a full first-order theorem prover. The prover can be extended along the lines of hyper-resolution. Another way is to modify the input clauses to be all Horn clauses or all-positive clauses and use almost the same strategy.

Chapter 5

A Finite-Model Finder

5.1 Introduction

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. This includes many decidable theories of interest, such as many of the description logics that are useful in knowledge representation as well as decidable subclasses of first-order logic based on the form of the quantifier prefix. If there is a model with a finite domain, then one can search for it in an exhaustive manner and verify that it is a model. Therefore, by searching for finite models and also using a theorem prover for proofs of unsatisfiability, we obtain a decision procedure for many decidable sublanguages of first-order logic (those having the finite model property). In addition, we obtain a useful tool that can sometimes demonstrate the satisfiability of sets of clauses that do not lie in any of the specialized languages.

There are a considerable number of complete theorem provers varying in efficiency and capability. However, on the other hand very little effort seems to have gone into finding whether a set of formulas is satisfiable. The primary reason is probably the nature of the problem itself— undecidable in the general case. Recent developments have brought out some diverse strategies. Most strategies for first-order logic exhaustively search for models. FINDER[Sla94] is a highly efficient model finder that performs an exhaustive search using contraint processing techniques. It has been used to discover several new results in quasigroups. Modgen[KZ94] is a theorem prover for finite domains that uses clause transformation. Given a domain, ModGen transforms first-order clauses to propositional clauses in a fashion that preserves satisfiability over that domain. Tammet[FLTZ93] and

Fermuller[FL94] describe model-finding strategies for several sublanguages of first-order logic.

In this chapter, we describe model-finder that is similar to ModGen. We describe the algorithm in the next section, and an alternate clause transformation. We also present results of comparing it with a model-finder for a subclass of first-order logic described in [Tam90].

5.2 Finite Model Finding Algorithm

To provide a meaning to sentences in first-order logic, some form of structure or interpretation is needed. Recall that an interpretation is made up of a domain and an interpretation function. The interpretation function connects the vocabulary of the first-order sentences to the domain. More specifically, constant symbols are interpreted to elements in the domain; function symbols of some arity k are defined by interpreting each k-tuple of domain elements uniquely to some domain element. A predicate symbol of arity n is interpreted to some set of n-tuples of domain elements. A model for a first-order formula is an interpretation that satisfies the formula. Our objective in this section is to describe a scheme to construct a model with a finite domain for a set of first-order clauses. We refer to a model with a finite domain as a finite model.

Suppose we wish to check if some set of clauses S in first order logic is satisfiable with a finite domain of size n. We can translate S to a ground set $G(S, n)$ in such a fashion that $G(S, n)$ is propositionally satisfiable if and only if S has a model of size n. We view the finite-model finder to be made of two distinct parts— a clause translator and a propositional prover. The clause translator takes a set of first-order formulas and obtains a set of propositional clauses. The propositional clauses are then tested for satisfiability with a very fast Davis-Putnam procedure[ZS94]. If it is known that the formulas that have models, have small models, then we can start at some domain size, say 1, transform the formulas and test them for satisfiability. If the propositional clause set is satisfiable, then from its model it is possible to construct a model for the first-order set. If the propositional clause set is unsatisfiable, then there is no model for the first-order set with this domain size and the next domain size can be tried.

We define a *domain instance* of a first order formula C_i to be a propositional clause created by replacing every variable in C_i with an element from the domain. Therefore a

clause with v unique variables has v^n domain instances, where n is the domain size. A domain instance of an atom, and that of a function term (including constants, which are treated as functions without arguments) are similarly defined. For example if we had the clause:

$$\{P(f(X,Y),Z), \neg Q(c, g(Z))\}$$

and the domain $\{1,2\}$, then one domain instance of the clause would be:

$$\{P(f(1,2),1), \neg Q(c, g(1))\}$$

A domain instance of $P(f(X,Y),Z)$ would be $P(f(1,2),1)$, and one of $f(X,Y)$ would be $f(1,1)$.

Given a set of first order formulas $S = \{C_0, C_1, \ldots, C_k\}$, and a domain $D = \{b_1, \ldots, b_n\}$, we show how to compute the set of propositional clauses $G(S,n)$ such that $G(S,n)$ is satisfiable if and only if S has a model of size n. $G(S,n)$ has four kinds of clauses.

- Function Interpretation Clauses

  These clauses are added to obtain an interpretation for the functions appearing in S. For every function f of arity k appearing in S we add the following clauses:

  $$f(d_{i1}, \ldots, d_{ik}) = b_1 \vee \ldots \vee f(d_{i1}, \ldots, d_{ik}) = b_n$$

  These clauses express the completeness of the function definition. The d_{ij} are chosen from $D = \{b_1, \ldots, b_n\}$ in all possible ways.

  We also add the following clauses which constrain the function definition to be single-valued.

  $$\begin{aligned}
  &f(d_{i1}, \ldots, d_{ik}) \neq b_1 \vee f(d_{i1}, \ldots, d_{ik}) \neq b_2 \\
  &f(d_{i1}, \ldots, d_{ik}) \neq b_1 \vee f(d_{i1}, \ldots, d_{ik}) \neq b_3 \\
  &\qquad\qquad \vdots \\
  &f(d_{i1}, \ldots, d_{ik}) \neq b_{n-2} \vee f(d_{i1}, \ldots, d_{ik}) \neq b_n \\
  &f(d_{i1}, \ldots, d_{ik}) \neq b_{n-1} \vee f(d_{i1}, \ldots, d_{ik}) \neq b_n
  \end{aligned}$$

$f(d_{i1},\ldots,d_{ik}) = b_1$ corresponds to $f(d_{i1},\ldots,d_{ik})$ being interpreted to b_1. Similarly $f(d_{i1},\ldots,d_{ik}) \neq b_1$ means that $f(d_{i1},\ldots,d_{ik})$ is not interpreted to b_1. Naturally, the literals are complements of each other.

- Universal Quantification Clauses

  If S is satisfiable, and has a model with a domain D, then, from the definition of universal quantification, the set of the domain instances of all the clauses is satisfiable as well. We call this set S_{DI}, which is a part of $G(S,n)$. The literals of the clauses in this set are domain instances of the atoms appearing in S.

- Predicate Interpretation Clauses

  For every literal $P(t_1,\ldots,t_k)$ appearing in the above computed domain instances of clauses S_{DI} we add the following propositional clauses:

$$P(d_{i1},\ldots,d_{ik}) \wedge t_1 = d_{i1} \wedge,\ldots,\wedge t_k = d_{ik} \Rightarrow P(t_1,\ldots,t_k)$$
$$\neg P(d_{i1},\ldots,d_{ik}) \wedge t_1 = d_{i1} \wedge,\ldots,\wedge t_k = d_{ik} \Rightarrow \neg P(t_1,\ldots,t_k)$$

  where $d_{i1},\ldots,d_{ik}$ are chosen from D in all possible ways. These clauses correspond to evaluating a domain instance under an interpretation. The ground literal $t_i = d_{ki}$ can be viewed as the term t_i being interpreted to the domain element d_{ki}. Sometimes the domain instance of an atom may contain elements of the domain as arguments. That is, t_m could itself be a domain element. In that case, we may view t_m as already interpreted to some domain element, and the literals of the form $t_m = d_{ij}$ are not included in the clauses. This is to eliminate vacuous implications and thus reduce the number of clauses generated.

- Subterm Interpretation Clauses

  These are similar to predicate interpretation clauses. For every domain instance of a function term $f(t_1,...t_k)$ appearing in C_{DI} we add the following propositional clauses:

$$f(d_{i1},\ldots,d_{ik}) = d_1 \wedge t_1 = d_{i1} \wedge \ldots \wedge t_k = d_{ik} \Rightarrow f(t_1,\ldots,t_k) = d_1$$
$$\vdots$$
$$f(d_{i1},\ldots,d_{ik}) = d_n \wedge t_1 = d_{i1} \wedge \ldots \wedge t_k = d_{ik} \Rightarrow f(t_1,\ldots,t_k) = d_n$$

where $d_{i1}, \ldots, d_{ik}$ and are chosen from D in all possible ways. These clauses correspond to evaluating a function term under an interpretation. As described for the Predicate Interpretation Clauses, if function arguments contain domain elements then the literals corresponding to them are not included.

Lemma 5.2.1 *$G(S, n)$ is satisfiable if and only if S is satisfiable in a model of size n.*

Proof. Suppose S is satisfiable, and has a model M, with domain $D = \{d_1, \ldots, d_n\}$. From the interpretation of the functions in M, we can set the corresponding literals in the Function Interpretation Clauses to be true. Since the function terms are uniquely interpreted in the model, the Function Interpretation clauses are satisfied. The Predicate Interpretation Clauses and Subterm Interpretation Clauses correspond to evaluating a predicate or a subterm under an interpretation, and so they are satisfied as well. By the definition of universal quantification, we can know that Universal Quantification Clauses are satisfied. Therefore $G(S, n)$ is satisfied.

Suppose $G(S, n)$ is satisfiable. Then it has a model M_g. We construct a model M, with finite domain $D = \{d_1, \ldots, d_n\}$, for S from M_g. M_g satisfies exactly one literal of the form $f(d_{i1}, \ldots, d_{ik}) = d_v$ for each f in S and all $d_{i1}, \ldots, d_{ik}$ in D; we can take d_v to be the interpretation of $f(d_{i1}, \ldots, d_{ik})$ in M. Similarly we can extend this to interpret all subterms, using the Subterm Interpretation Clauses. Each of the domain instances of the clauses has at least one literal satisfied by M_g. If M_g satisfies $P(t_1, \ldots, t_k)$, then by the Predicate Interpretation Clauses it also satisfies $P(d_{i1}, \ldots, d_{ik})$ where d_{ij} is the interpretation of t_j in M. M satisfies $P(d_{i1}, \ldots, d_{ik})$ and also $P(t_1, \ldots, t_k)$. So M is a model of S. □

Given the above translation, the outline of the model finding algorithm is given in Figure 5.2. The algorithm does not terminate if there is no finite model.

There are some other clauses that are also added to $G(S, n)$ before testing it for satisfiability. To define equality, we add the unit clauses $\{d_i = d_i\}$ for all i and $\{ d_i \neq d_j\}$ for all different i and j. This provides an extensional definition for equality. Clearly, these additions only change the satisfiability of $G(S, n)$, by allowing the usual meaning of equality.

We illustrate the working of the model-finder with the following simple example. This example also illustrates the growth in the number of propositional clauses with increasing domain size.

Algorithm 5 (Computing a Finite Model for a set of First-Order Formulas)
Input: [A Satisfiable Clause Set S that has a finite model]
Output: [A finite model for S]

```
1   procedure find_finite_model(S)

2   begin
3      D ← {}
4      n ← 0
5      done ← false
6      while done = false do
7         n = n + 1;
8         D ← D ∪n
9         compute G(S, n)
10        if G(S, n) is satisfiable with model M_g then
11           M ← transform(M_g)
12           done ← true
13        endif
14     endwhile
15     return M

16  end
```

Figure 5.1: Finite-Model finding Algorithm

Example. Consider the following set S of clauses:

$\{\neg p(x, x)\}$
$\{p(x, f(x))\}$

First, a domain of size one is tried. $G(S, 1)$ contains the following propositional clauses:

The Universal quantification clauses or domain instances:
1. $\{\neg p(1, 1)\}$
2. $\{p(1, f(1))\}$

The function interpretation clauses:
3. $\{f(1) = 1\}$

The predicate interpretation clauses:
4. $\{\neg p(1, 1), \neg(f(1) = 1), p(1, f(1))\}$ 5. $\{p(1, 1), \neg(f(1) = 1), \neg p(1, f(1))\}$

From clauses 1,2,3 and 5 it can be seen that $G(S, 1)$ is unsatisfiable. The next domain size is tried. $G(S, 2)$ contains the following propositional clauses:

The Universal quantification clauses or domain instances:
1. $\{\neg p(1, 1)\}$
2. $\{p(1, f(1))\}$
3. $\{\neg p(1, 2)\}$
4. $\{p(1, f(2))\}$
5. $\{\neg p(2, 1)\}$
6. $\{p(2, f(1))\}$
7. $\{\neg p(2, 2)\}$
8. $\{p(2, f(2))\}$

The function interpretation clauses:
9. $\{f(1) = 1, f(1) = 2\}$
10. $\{f(2) = 1, f(2) = 2\}$

11. $\{\neg(f(1) = 1), \neg(f(1) = 2)\}$
12. $\{\neg(f(2) = 1), \neg(f(2) = 2)\}$

The predicate interpretation clauses:

13. $\{\neg p(1,1), \neg(f(1) = 1), p(1, f(1))\}$
14. $\{p(1,1), \neg(f(1) = 1), \neg p(1, f(1))\}$
15. $\{\neg p(1,2), \neg(f(1) = 2), p(1, f(1))\}$
16. $\{p(1,2), \neg(f(1) = 2), \neg p(1, f(1))\}$
17. $\{\neg p(2,1), \neg(f(1) = 1), p(2, f(1))\}$
18. $\{p(2,1), \neg(f(1) = 1), \neg p(2, f(1))\}$
19. $\{\neg p(2,2), \neg(f(1) = 2), p(2, f(1))\}$
20. $\{p(2,2), \neg(f(1) = 2), \neg p(2, f(1))\}$
21. $\{\neg p(1,1), \neg(f(2) = 1), p(1, f(2))\}$
22. $\{p(1,1), \neg(f(2) = 1), \neg p(1, f(2))\}$
23. $\{\neg p(1,2), \neg(f(2) = 2), p(1, f(2))\}$
24. $\{p(1,2), \neg(f(2) = 2), \neg p(1, f(2))\}$
25. $\{\neg p(2,1), \neg(f(2) = 1), p(2, f(2))\}$
26. $\{p(2,1), \neg(f(2) = 1), \neg p(2, f(2))\}$
27. $\{\neg p(2,2), \neg(f(2) = 2), p(2, f(2))\}$
28. $\{p(2,2), \neg(f(2) = 2), \neg p(2, f(2))\}$

$G(S,2)$ is satisfiable and has the following model, showing only the positive literals
$\{p(2,1), p(1,2), f(2) = 1, f(1) = 2, p(2, f(2)), p(1, f(1))\}$
From this model the following first-order model (D, I) is obtained:
$D = \{1, 2\}$
$f^I : 1 \mapsto 2$
$f^I : 2 \mapsto 1$
$p^I is \{< 1,2 >, < 2,1 >\}$
□

Problem	Prefix	Tammet	Model Size	FMFinder	Model Size
2.2	∀∃	Y	2	Y	1
2.3	∃∃∃∀	Y	1	Y	1
3.1	∃∀∃∃	Y	2	Y	2
4.1	∃∀	Y	1	Y	1
9.1	∀∃	Y	2	Y	2
14.1	∀∀∃	Y	7	Y	6
14.2	∀∀∃	—	—	—	—
14.7	∀∀∃	—	—	—	—
15.2	∃∃∀∃	Y	2	Y	1
15.7	∃∀∃∀	Y	2	Y	1
16.3	∃∀∀∃	Y	2	Y	1
17.4	∀∃∀∃	—	—	Y	2
18.3	∃∃∀∀∃	Y	4	Y	3

Table 5.1: Experiments with the Model Finder

5.3 Performance

Church[Chu56] presents a large set of formulas in several decidable subclasses of first-order logic. Some of the formulas are satisfiable. The Table 5.1 presents the results of running some of the satisfiable problems on the model-finder. The results are compared with Tammet's specialzed method for finding models thesis[Tam90]. Each row of the table contains the problem number from Church's book, the quantifier prefix associated with the formula, whether the system found a model and the size of the domain of the model found. Our model finder did not take more than a few seconds for any problem.

It is interesting to see that our model finder despite the number if propositional clauses it generates for large domains compares well with the specialized technique for problems of this class. Further, in most cases our system found smaller models. In one case the specialized technique did not find a model of size 2. This puzzles us.

Chapter 6

Description Logics

6.1 Introduction

Description Logic Systems [1] [BH91, Mac91, PSMB+91, Pet91, BS85] provide a means for representing knowledge using *concepts* and *roles*. Although there is no single definitive description logic, many of the prevalent systems are descended from KL-ONE[BS85]. They are generally made of two distinguishable components. One provides the user with a formalism to represent knowledge in an abstract sense— a provision to represent relationships between concepts and roles in a general sense. The other allows the user to concretize this abstraction by allowing extensional instantiations of the general definitions. Retrieval of information is usually based on some deductive processes involving both of these components. Description logics are motivated by the search for expressive knowledge representation languages that also allow computationally decidable reasoning faculties. This has resulted in differing approaches to building systems: Some systems[BH91] have relatively rich formalisms to express concept definitions but suffer the possibility of having to deal with intractable and sometimes even undecidable problems sometimes causing the reasoner to not terminate. In fact [BH91] is a semantic-tableaux theorem prover for description logics. Some provide extended expressivity[Mac91] but have reasoners that are incomplete for even decidable problems. Another kind[PSMB+91] prefer to provide a small and compact language for which reasoning is for the most part complete and efficient.

Buchheit et al[BDS93] point out that description logic systems should provide mechanized methods to perform at least the following tasks: To check whether a knowledge-

[1] Also known as Terminological Knowledge Representation Systems or Concept Language Systems

base is consistent— that is whether there exists a model for the knowledge-base; Determining whether a concept is satisfiable— whether there exists some model in which the concept is non-empty; Determining whether a concept is more general than, or subsumes, another; To check whether a particular individual is an instance of a concept — if every interpretation of the concept contains the individual. Of these, determining concept subsumption is the most fundamental task [BPS94, HN90]. The subsumption relation defines a partial ordering over concepts. Usually, description logic systems are equipped with the capacity to construct this partial ordering involving all the defined concepts. This process is called *classification.*

Heinsohn et al [HKNP92] describe an empirical analysis of the following description logic systems— BACK, CLASSIC, KRIS, LOOM, MESON and SB-ONE. The systems are compared for features and expressivity offered by the language, degree of inferential completeness, and classification speed. The tests for inferential completeness, by no means exhaustive, look for seemingly obvious conclusions based on a few concept definitions. They also include some known pathological cases for existing systems. Classification speeds are measured for some realistic and some randomly generated knowledge-bases. The test examples are described in "a common terminological language". This is an intermediate language that is derived from the description logics compared. This language corresponds to a subset of first-order logic, and any concept or role definition in this language is easily translated to sentences in first-order logic.

This chapter examines the performance of a reasoner made up of a first-order theorem prover combined with a finite-model finder when applied to the problem of detecting subsumptions between concepts in description logic systems. Theoretically, this combination can be used as a decision procedure for any finitely controllable subclass (ie if any model exists, a finite model must exist) of first-order logic. The logic is then said to have the finite model property. The underlying logic of many description logics has the finite model property. The idea of using a theorem prover with a model finder for performing inferences in description logic systems is not new. Hollunder and Nutt [HN90] mention it. Tammet [FLTZ93] describes how a resolution-based theorem prover and model finder, for a restricted form of predicate calculus, can be applied toward subsumption checking. However, the description logic considered is quite limited and the test example considered is very trivial. So far such approaches have been dismissed because standard theorem proving techniques have not been thought to be fast enough. We demonstrate that this is not the

case using tests developed in [HKNP92]. The completeness of the theorem prover and the finite-model finder guarantees that the system either determines that a concept subsumes another, or presents a model that counters the subsumption. For the system to perform efficiently we have developed a number of preprocessing steps. We believe that the richness in expressivity that this reasoner allows, without compromising efficiency, makes it a promising mechanism for discovering subsumptions in description logic systems.

We examine how RRTP[PP97a] augmented with a finite-model finder compares in performance with the description logic systems tested in [HKNP92]. We restrict our attention to the tests involving subsumptions. This includes the inference tests, and the classification for the realistic knowledge-bases. RRTP obtains "replacement rules" from the input clauses and replaces ground terms by their predicate definitions as dictated by the replacement rules. In this fashion ground instances are collected, and are periodically tested for unsatisfiability. The prover has been shown to be complete.

We have developed some preprocessing techniques that makes our system efficient. For subsumption checking we select only clauses relevant to the proof. This is often a small subset of the clause set that represents the knowledge-base. Classification poses a different problem because determining subsumptions in a pairwise fashion is very tedious and impractical. For classification we use the finite-model finder to eliminate checking several subsumptions. It should also be noted that the model sizes observed were consistently low, permitting the finite-model finder to detect them very quickly.

The rest of the chapter is structured as follows: Computing relevant clauses for subsumption computation, and tests for inferential completeness are discussed in section 3. Classification techniques and test results are described in section 4. In section 5 we discuss possible improvements. The appendix outlines a description of the tests.

6.2 Comparison of Inferential Abilities

In this section we discuss how subsumption checking is done using RRTP. We also discuss some tests designed to probe the inferential abilities of description logic reasoners. They have been used before to compare some description logics [HKNP92]. Although these tests are not a comprehensive suite, they are varied and take into account several language constructs offered by description logics. We were also interested in the performance of some other theorem provers on these tests, and we ran these tests on OTTER[McC90],

CLIN[LP92, Lee90b] and SPRFN[Pla88b] as well. We compared their performance with those of the the description logics based on reports in [HKNP92]. It is interesting to note that not all theorem provers were able to derive all of the inferences described by these tests.

6.2.1 Subsumption Checking in First-Order Clause Sets

Concepts are used to represent classes of objects in some domain of interest. Roles represent binary relations between objects. Starting from basic or atomic concepts and roles, complex concept and role descriptions may be defined. Although the specifics of such concept-forming (or role-forming) constructs vary across description logics, most of them include conjunction, complementation and quantification over roles. Consider, for example, the definition of the concept PARENT from [BH91]

PERSON $\sqcap$ $\exists$child:PERSON

This defines the class of objects that are formed by the intersection of the following: objects defined by PERSON; and those objects for which some child role is filled by an object defined, again by PERSON. Number restrictions on roles are also commonly available constructs for building concepts. For example, PARENT may be defined using number restriction on the role child as follows.

PERSON $\sqcap$ $\geqslant$ 1 child:PERSON

Description Logic systems are usually composed of two separate components— an assertional part and a terminological part. The T-Box, the terminological part, provides the user with a means to define classes and relations in a general sense. The A-box, the assertional part, allows the user to assert information about the domain. The A-Box and the T-Box together constitute the knowledge-base.

For example, the T-Box may have the following concept definitions [BH91]

PARENT $\sqcap$ gender:Male

PARENT $\sqcap$ gender:Female

which define the concepts FATHER and MOTHER, respectively.

The A-Box may contain the following assertions:

```
PERSON(James)        gender(James,Male)
PERSON(Susan)        gender(Susan,Female)
child(James,Susan)
```

From a logical perspective, concepts can be viewed as unary predicates, and roles as binary predicates. Concept definitions may be easily translated to sentences in first-order logic maintaining the intended semantics of the concept-forming constructs. For example, the first definition for PARENT would correspond to the following sentence:

$$\forall x\ parent(x) \Leftrightarrow (person(x) \wedge \exists y\ (child(x,y) \wedge person(y)))$$

Concepts are interpreted as subsets of some domain of interest, and roles as binary relations over the domain. Formally [BDS93], an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$ — the domain of $\mathcal{I}$ and an interpretation function $.^{\mathcal{I}}$. The interpretation function $.^{\mathcal{I}}$ maps concepts to be subsets of Δ and roles to be subsets of $\Delta \times \Delta$. A concept is satisfiable provided some interpretation maps it to a non-empty set. To determine if C_1 subsumes C_2 it is sufficient to examine if the concept defined by $C_2 \sqcap \neg C_1$ is unsatisfiable. The problem of determining subsumptions can be converted to one of determining the unsatisfiability of a first-order clause set.

The concepts defined in the T-Box is first translated to an equivalent logical description— a set of skolemized first-order clauses which we shall refer to as $TBox$. We illustrate this with an example. Consider the T-Box containing the single concept definition for PARENT

PARENT $\equiv$ PERSON $\sqcap$ $\exists$child:PERSON

The logical description of the definition is

$$\forall x\ parent(x) \Leftrightarrow (person(x) \wedge \exists y\ (child(x,y) \wedge person(y)))$$

From this we get $Tbox$— the conjunction of the following four skolemized first-order clauses.

```
{parent(X), not(person(X)), not(child(X,Y)), not(person(Y))}
{person(X), not(parent(X))}
{child(X,f(X)), not(parent(X))}
{person(f(X)), not(parent(X))}
```

Each clause is a disjunction of literals. For example, `parent(X)` and `child(X,Y)` are literals. The function symbol `f` is a skolem function symbol, which is uniquely constructed for this concept definition. Note that all the variables are universally quantified. We refer to a predicate symbol that corresponds to a concept, as a concept for the sake of brevity. For example, the predicate symbol `parent` corresponding to the concept PARENT is referred to as a concept.

Suppose that a T-Box definition includes concepts C_i and C_j. For C_i to subsume C_j, in all interpretations $\mathcal{I}$, $C_j^{\mathcal{I}} \subseteq C_i^{\mathcal{I}}$. Therefore, logically, C_i subsumes C_j if and only $Tbox \Rightarrow \forall x(C_j(x) \Rightarrow C_i(x))$. This is valid if and only if the set of clauses $Tbox \wedge \{C_j(c)\} \wedge \{\neg C_i(c)\}$ is unsatisfiable.[2] Therefore, to determine the validity of the subsumption, the refutation theorem prover checks if $Tbox \wedge \{C_j(c)\} \wedge \{\neg C_i(c)\}$ is unsatisfiable. However, if it is satisfiable, and has a finite model then the finite-model finder will find it and negate the subsumption.

Instead of simply running the theorem prover and the finite-model finder on the clause set, we make a few observations that make subsumption detection more efficient.

Definition 6.2.1 *Suppose we have a set of clauses S. Then a clause C in S is fully matched if for every literal L in C there exists a clause D in S and a literal M in D such that L and the complement of M unify.*

Definition 6.2.2 *Suppose a clause set S is unsatisfiable. A clause C in S is relevant to the unsatisfiability of S if S - { C }is satisfiable.*

Lemma 6.2.1 *Suppose we have a set of clauses S that is unsatisfiable. A clause C in S that is not fully matched is not relevant to the unsatisfiability of S.*

Proof. Assume that C is relevant. Then $S - \{C\}$ is satisfiable and has a model. Let L be the unmatched literal in C. Without loss of generality assume that L is positive. Since no clause in $S - \{C\}$ contains a literal that unifies with the complement of L, we can set L to be identically true in the model of $S - \{C\}$. Suppose this model no longer modeled some clause; then that clause must have contained a literal that could have unified with the complement of L. Thus, the model now satisfies S, contradicting our position. Therefore C cannot be relevant. □

Definition 6.2.3 *An alternating path involving clauses C_1 and C_n is a sequence (L_1, C_1), (M_1, C_1), (L_2, C_2), $(M_2, C_2), \ldots, (L_n, C_n)$ where L_i and M_i are literals in C_i and $L_i \neq M_i$ for all i. Also, for all $i < n$, M_i and the complement of L_{i+1} unify.*

An unsatisfiable set of clauses where every clause is relevant is called minimum unsatisfiable.

[2]Note that the symbol c is a skolem constant— a skolem function of arity zero, introduced to replace the outermost existentially quantified variable in a quantifier prefix.

Tests $1a - 1d$ are straightforward. These tests involve showing that the conjunction of disjoint concepts result in inconsistent concepts. The tests vary in the way the disjoint nature of two concepts is expressed. As anticipated, the theorem provers had no difficulty in drawing the correct inference in any of the cases. It is interesting to note that $1d$, poses a very trivial inference test when represented in clause form. Furthermore, all of these examples took insignificant amounts of time for RRTP. All of them were under a few seconds. Tests $2a - b$ involve detection of inconsistent concept definitions given incompatible value restrictions. While $2a$ is a straightforward manifestation of this, $2b$ brings in disjoint concepts. Tests $3a - 3e$ further extend the idea of $2a$ with complicated descriptions that combined disjoint concepts with limiting the range of a role and value restrictions. It is interesting to note that even some theorem provers were not able to derive the inferences in these tests. RRTP derived all the inferences, and most of them in reasonable time— lesser than ten seconds. Tests $4a - 4b$ use range restriction on roles to test for some form of case-based reasoning. This construct is not available in all the description logics. The languages that have the construct, however, handle it incompletely. Test 5 uses a construct that can cause undecidability in subsumption. Test 6 tests equality reasoning over attributes. Attributes are like roles but they define functions over the domain. Test 7 is a simple test for inverse as a concept-forming operator. The results of these tests are tabulated in Table 6.1. We note that in terms of deriving inferences, theorem provers in general were better than the description logic systems but not necessarily complete. RRTP turned out to be the most complete with respect to these tests. Note— current versions of some of the terminological systems may report better performances.

Hard Inferences

There were four categories of tests involving the "hard cases". Concepts may be hierarchically defined, leading to a definition tree. If two concepts have such tree-like definitions, then to compute the subsumption relation between them, some description logic systems often naively expand their definition. This may lead to exponential increase in the length of the definition. The first three categories involve similar independent definitions for a set of concepts $C_0, \ldots, C_n$ and $D_0, \ldots, D_n$. The first category involves concepts defined as conjuncts of qualified role restrictions in a hierarchical fashion.

$C_{i+1} \equiv \forall\ R_1 : C_i \sqcap \forall\ R_2 : C_i$ and
$D_{i+1} \equiv \forall\ R_1 : D_i \sqcap \forall\ R_2 : D_i$ and for i ranging from 0 to n.

For different values of n it is tested if D_n subsumes C_n. If the definition for C_{i+1} (similarly D_{i+1}) is expanded by replacing occurrences of C_i with C_i's definition and so on, then we get definitions for C_n and D_n that are exponential in n. This is avoidable while computing subsumption, but some terminological systems incur the exponential blow up in the definition.

The next category is similar to the first, but the concepts are defined as value restrictions placed on conjuncts of other concepts. That is, we have for C_{2i} and C_{2i+1}

$$C_{2i} \equiv \forall\ R_1 : (C_{2i-2} \sqcap C_{2i-1})$$
$$C_{2i+1} \equiv \forall\ R_2 : (C_{2i-2} \sqcap D_{2i-1})$$

In the third category of tests, the concepts are defined in such a fashion that expansions of definitions are necessary to compute subsumptions, and thus even the best algorithms have no recourse but to expand the definition. Not surprisingly, it is in this example that RRTP (with the finite model-finder) did substantially better than the other description logic systems, as no such expansion of definitions is necessary. Instead, to test if a concept C_n does not subsume a concept D_n, a model is sought for the clauses defining the concepts with two unit clauses $\{\neg C_n(c)\}$ and $\{D_n(c)\}$. Since in all of the above examples the concepts C_i and D_i are unrelated, the finite model-finder took very little time in finding a model contradicting the subsumption in each case. The domain size of this model was one. Even in general, it was interesting to note that absence of subsumptions were detected quickly in most cases because the model sizes were usually very small. This fact is exploited in the classification process described in the next section. We note that times indicated, include the time taken to perform clause translation.

The last of these tests were based upon the result that term-subsumption in description logics that include conjunction, value restrictions and qualified existential restriction is NP-hard. However, only two terminological systems could express this case, and even among those KRIS alone successfully drew all the inferences. LOOM, despite its speed, apparently failed to draw the necessary inference. RRTP failed to draw two necessary inferences, and took a lot of time drawing another.

6.3 Classification

Recall that a concept C_j is said to subsume a concept C_i, given a T-Box definition $Tbox$, provided $Tbox \Rightarrow \forall x(\ C_i(x) \Rightarrow C_j(x)\ )$. Given a T-Box, the computation of the subsumption hierarchy of concepts is termed as classification. Classification involves computing several subsumptions— all the subsumptions that exist between concepts defined in the knowledge-base. Subsumption is a transitive relation. It is easy to see that if a concept C subsumes C', and C' subsumes D, then C subsumes D. That is, $Tbox \Rightarrow ((\forall x(D(x) \Rightarrow C'(x)) \wedge \forall x(C'(x) \Rightarrow C(x))) \Rightarrow (Tbox \Rightarrow \forall x(D(x) \Rightarrow C(x))))$ Usually the subsumption hierarchy that is computed is the smallest relation whose reflexive, transitive closure defines the subsumption relation between concepts. This relation is termed as *immediate subsumption*. A concept C immediately subsumes another concept D if C subsumes D and there exists no other concept C' such that C subsumes C', and C' subsumes D. No concept immediately subsumes itself.

Generally description logic systems classify concepts in an incremental fashion. When a concept is defined, it is inserted into the hierarchy built so far, based on its immediate predecessors— the concepts that immediately subsume it; and its immediate successors— the concepts that are immediately subsumed by it. These predecessors and successors are determined by performing subsumption tests. Some sophisticated ordering traversal techniques are usually used to reduce the number of subsumption tests. In some cases— for example if all the concepts in the knowledge-base are unrelated— classification can result in subsumption comparisons between all possible pairs of concepts. This is quadratic in the number of concepts. Heinsohn et al [HKNP92] conjecture that the number of tests is quadratic in the average case as well. Since subsumption determination is an expensive operation, Heinsohn et al stress that the key issue in designing efficient classification algorithms is minimizing the number of subsumption tests. Motivated by this, a strategy for eliminating several non-existent subsumptions very quickly is described.

This classification technique is novel in that the subsumption relation, which we refer to as *subsume*, is not incrementally built when concepts are being added to the T-Box. Instead, using the set of formulas that represent the T-Box and the concepts contained in it, a supra-relation of the relation *subsume* is computed. We refer to the relation defined by this supra-relation as *possible*. The relation *possible* is then suitably pruned, to yield *subsume*, by checking whether its members denote valid subsumptions or not. The relation

possible is simply a weaker form of the subsumption relation. Suppose that all models of a concept C with a domain of size k, satisfy $Tbox \Rightarrow \forall x(D(x) \Rightarrow C(x))$, we say C possibly subsumes D or $(C, D) \in$ *possible*. It is straightforward to see that *subsume* $\subseteq$ *possible*.

Many subsumptions can be detected by simply inspecting the clauses. For example, a binary clause may indicate a logical implication between two concepts. Similar techniques to detect obvious subsumptions are present in most description logic systems. A relation *partial* is partially built from these obvious subsumptions. Obviously *partial* $\subseteq$ *subsume*. Using *possible* and *partial* and the subsumption algorithm described in Figure 6.1 we compute *subsume*. We describe two strategies that exploit the transitive nature of the relation to minimize the total number of subsumptions to be made.

Four realistic knowledge-bases were classified using these methods. The times to perform the classifications are compared with some description logic systems' performances. The times are quite satisfactory. However, what was more impressive was the fact that these methods reduce the number of subsumption tests to be made by several orders of magnitude.

6.3.1 Eliminating Subsumption Tests by Model Generation

We discuss the construction of the relation *possible*. Given any set of clauses $Tbox$ representing some T-Box and some concepts C_1 and C_2 defined in it, suppose that we have an interpretation $\mathcal{I}$ such that $d \in C_1^{\mathcal{I}}$, but $d \notin C_2^{\mathcal{I}}$. Then, clearly C_2 does not subsume C_1, because such an interpretation invalidates $Tbox \Rightarrow \forall x(C_1(x) \Rightarrow C_2(x))$. This observation motivates a strategy for eliminating unnecessary subsumption checking between several unrelated concepts.

Recall from Section 2 that $G(S, n)$ is a propositional set of clauses obtained by transforming S. $G(S, n)$ has a propositional model— M_g, if and only if S has a first-order model M with a domain of size n. Also if M_g maps some literal $P(d)$ to true, then M interprets the predicate P to some set containing d. Now suppose that for some value of n, we construct the propositional clause set $G(Tbox, n)$ and add to the set the unit clause $\{ C_i(a) \}$. We refer to this propositional set as $ground\_set(Tbox, C_i, n, a)$. The constant a is chosen from the domain used to construct $G(Tbox, n)$. If $ground\_set(Tbox, C_i, n, a)$ is satisfiable and has a model, then there exists a first-order model $\mathcal{I}$ that satisfies C_i, and because of the unit clause $\{ C_i(a) \}$, $a \in C_i^{\mathcal{I}}$. Furthermore, for any concept C that subsumes the concept C_i, $a \in C^{\mathcal{I}}$. It follows that for C to subsume C_i, any propositional model for

Algorithm 7 (***make_possible*: Compute the relation *possible***)
Input: [TBox, Smallest model to be tried]
Output: [The relation *possible*]

```
1   Concept ← concepts occurring in Tbox
2   n ← k
3   possible ← φ
4   repeat
5      n ← n + 1;
6      compute G(Tbox, n);
7   until G(Tbox, n) is satisfiable
8   forall C_i ∈ Concept do
9      ground_set(Tbox, C_i, n, a) ← G(Tbox, n) ∪ {C_i(a)}
10     compute unit consequences U for
11     ground_set(Tbox, C_i, n, a)
12     if C(a) ∈ U then
13        possible ← possible ∪ (C, C_i)
14     endif
15  enddo
```

Figure 6.2: Constructing the relation *possible*

$ground\_set(Tbox, C_i, n, a)$ must interpret the concept literal $C(a)$ to be true. In other words

$ground\_set(Tbox, C_i, n, a) \vdash C(a)$.

If the concept literal $C(a)$ is not a logical consequence of $ground\_set(Tbox, C_i, n, a)$ then the concept C does not subsume C_i.

By computing the unit logical consequences of $ground\_set(Tbox, C_i, n, a)$ the concepts that may possibly subsume C_i can be determined. Based on this we construct the relation *possible*, which is simply the subsumption relation defined for domains of size n. That is, if $(C_2, C_1) \in possible$, then any model for C_1 of size n satisfies $Tbox \Rightarrow \forall x(C_1(x) \Rightarrow C_2(x))$. The algorithm in Figure 6.2 describes the construction of *possible*. For every concept C_i defined in the $Tbox$, the unit consequences of $ground\_set(Tbox, C_i, n, a)$ are computed. Sometimes, $ground\_set(Tbox, C_i, n, a)$ is unsatisfiable, and so all of the concept literals are unit consequences. For such cases a higher value of n can be tried to recompute $ground\_set(Tbox, C_i, n, a)$. We use k to decide how small the domain we consider should be. Computing unit consequences of propositional clause sets is described in [CP92].

The technique to compute unit consequences of a propositional set has not been incorporated yet. However, the finite-model finder has been used to accomplish the same objective. To find the concepts in S that subsume C_i, the finite-model finder is used to create some model M for $ground\_set(Tbox, C_i, n, a)$. Only those concepts C_j such that $M \models C_j(d)$ can subsume C_i. This is a much weaker way to construct *possible*. To make it stronger, for each such C_j we can try to find a model M_{ij} such that $M_{ij} \models ground\_set(Tbox, C_i, n, a) \wedge \{\neg C_j(a)\}$. If such an M_{ij} exists then $C_j(a)$ is not a unit logical consequence of of $ground\_set(Tbox, C_i, n, a)$ and cannot subsume C_i. Otherwise, C_j possibly subsumes C_i, and $(C_j, C_i) \in possible$. Although the result of this computation is the same as computing unit consequences, this strategy is less efficient.

6.3.2 Determining Subsumptions Using the Theorem Prover

The first round of subsumptions are detected by inspecting the clauses that logically describe the T-Box. Suppose we have a clause of the form $\{C(X), \neg D(X)\}$; it is easy to see that C subsumes D. These obvious subsumptions are then extended by transitivity. Most description logic systems use similar syntactic methods to detect obvious subsumptions. In [BHN+92], C is said to be a *told subsumer* of D, if the subsumption is readily apparent from the definition of D. We construct the relation *partial* by obtaining all the told subsumers of the concepts.

Using the relations *possible*, *partial*, and the subsumption testing algorithm described in Figure 6.1, the relation *subsume* is completely constructed. We refer to those concept pairs in *possible*, that are not present in *partial* as possible subsumptions. Basically, the subsumption testing can be used to check which of the possible subsumptions are actual subsumptions. A brute force way would be to test all possible subsumptions and retain them, or discard them depending on the result of the test. By exploiting the transitive nature of the subsumption relation, the total number of subsumption tests made is reduced considerably. We discuss two strategies described in Figure 6.3 to do this. The main idea in both of the strategies is to incrementally modify *partial* and prune *possible* until they converge to the same relation— which is the desired subsumption relation, *subsume*.

The algorithm *compute_subsume*1 uses the transitively closed ordering *possible* and the partially constructed *partial*, also transitively closed. The result of each subsumption test is propagated within the relation *partial*— if the test succeeds; and within

Algorithm 8 (Computing Subsumption Relation for all Concepts)
Input: [TBox, Subset and Superset of the subsumption relation]
Output: [Subsumption Relation for all Concepts]

```
 1  procedure  compute_subsume1(Tbox,partial,possible)
 2  while partial ≠ possible do
 3     pick some (C, D) ∈ possible - partial
 4     if test_subsume(Tbox, C, D)
 5        then mark_subsume(C,D)
 6     else unmark_possible(C,D)
 7     endif
 8  endwhile
 9  subsume ← partial
10  return subsume

 1  procedure unmark_possible(C,D)
 2  begin
 3     possible ← possible — (C, D)
 4     forall {X | (X, D) ∈partial}do
 5        unmark_possible(C, X)
 6     enddo
 7     forall {X | (C, X) ∈partial}do
 8        unmark_possible(X, D)
 9     enddo
10  end

 1  procedure mark_subsume(C,D)
 2  begin
 3     partial ← (C, D)∪ partial
 4     forall {X | (D, X) ∈partial} do
 5        mark_subsume(C, X)
 6     enddo
 7     forall {X | (X, C) ∈partial} do
 8        mark_subsume(X, D)
 9     enddo
10  end
```

Figure 6.3: Computing the Subsumption Relation

possible— if the test fails. The relation *partial* is updated by *mark_subsume*. The *possible* relation is pruned by *unmark_possible*. The sequence in which the possible subsumptions to be tested are picked can influence the performance. There is no specific sequence that will work well for all cases. One way is to pick possible subsumptions, that are transitive consequences of other possible subsumptions, only after (and if necessary) the antecedent subsumptions have been tested. This idea works well if many of the possible subsumptions are indeed valid subsumptions. Different methods of choosing the pair to test, have similar best and worst case scenarios. Note that this algorithm returns the entire subsumption relation— which is the transitive closure of the hierarchy.

In many examples we noticed that *possible* is not very different from *partial*. The algorithm *compute_subsume*2 does very well on such examples. Moreover the subsumption hierarchy based on immediate subsumption is returned, as opposed to the entire relation. First, *possible* and *partial* are first reduced to the smallest relations whose reflexive transitive closures are *possible* and *partial* respectively.

The difference between *compute_subsume*1 and *compute_subsume*2 stems from the fact that the transitive nature of the relations are made implicit in the latter. This results in more compact definitions of *possible* and *partial*. When a possible subsumption is tested, a positive test simply involves adding the corresponding concept pair to *partial*. A negative result, on the other hand, can cause the addition of several concept pairs to *possible*, all of which were hitherto implicit by transitivity. This is done by *update_possible*. The worst case of *compute_subsume*2 is when none of the possible subsumptions hold. This can result in a quadratic number of subsumption tests in the concepts. However, this technique works quite well in practice, as most possible subsumptions are actually subsumptions.

Since the clause sets corresponding to the knowledge-bases are fairly large, relevant clauses are selected to test for subsumptions. It is interesting to note that in almost all the subsumptions left to be tested the distance between the concepts tested for subsumption was less than 3, so in many cases the first iteration of the algorithm picked up all the relevant instances. The theorem prover's performance significantly improved when relevant clauses were used. Time to detect the subsumptions went down by a factor of 100 in some cases.

Algorithm 9 (Computing Subsumption Hierarchy for all Concepts)
Input: [TBox, Subset and Superset of the subsumption relation]
Output: [Subsumption Hierarchy for all Concepts]

```
 1  procedure  compute_subsume2(Tbox,partial,possible)
 2  begin
 3     while partial ≠ possible do
 4        pick some (C, D) ∈ possible - partial
 5        if test_subsume(Tbox, C, D)
 6           then partial ← partial ∪(C, D)
 7        else update_possible(C,D)
 8        endif
 9     endwhile
10     subsume ← partial
11     return subsume
12  end

 1  procedure update_possible(C,D)
 2  begin
 3     possible ← possible - (C, D)
 4     forall {X | (D, X) ∈partial}do
 5        possible ← possible ∪(C, X)
 6     enddo
 7     forall {X | (X, C) ∈partial}do
 8        possible ← possible ∪(X, D)
 9     enddo
10  end
```

Figure 6.4: Computing the Subsumption Hierarchy

6.3.3 Pruning the Clause Set and Reducing Subsumption Tests

Sometimes it is worthwhile to inspect the clause set and identify concepts that will not be relevant to part of the classification process. Such concepts and sometimes the clauses that involve them no longer need be considered. Reducing the number of clauses for the classification process is desirable because the performances of both the theorem prover and the model-finder improve. For example, knowledge-bases sometimes have several concept definitions each involving only one other concept. It is easily shown that concepts defined in such a manner are immediately subsumed only by the concepts defining them, and the clauses involving such concepts can be removed.

A concept that is not interpreted to any domain element in all models of $Tbox$ is an *empty* concept. For example `bot` described in Figure 6.5 is an empty concept. Vacuously, an empty concept is subsumed by all concepts. Similarly we have a *full* concept— a concept that is interpreted to every domain element in all models of $Tbox$ is a *full*. `top` is an example of a full concept. A full concept subsumes all other concepts. Suppose that in a $Tbox$ definition all the concept literals corresponding to some concept are negative, Then the concept is a *negative* concept. `man` and `woman` are examples of negative concepts. Similarly, if the concept literals are all positive, the concept is a *positive* concept. `human` is an example of a positive concept. Positive concepts are subsumed only by full concepts. This makes it unnecessary to consider positive concepts while computing the *possible* relation.

Suppose that a negative concept C is present in only one clause, and if such a clause is binary, say $\{ D(\mathrm{X}), \neg C(\mathrm{X}) \}$. Clearly D subsumes C. Furthermore, D is the only immediate subsumer of C. That is, any concept that subsumes C subsumes D as well. We show that if there exists any concept A, in $Tbox$, that does not subsume D, then A does not subsume C as well. Suppose A does not subsume D. Then there is an interpretation $\mathcal{I}$ that satisfies $Tbox \wedge \{\neg A(a)\} \wedge \{D(a)\}$. We modify $\mathcal{I}$ such that $a \in C^{\mathcal{I}}$. This is possible because C is present in only one clause and $\mathcal{I}$ satisfies that clause— $a \in D^{\mathcal{I}}$. So it is easy to construct $\mathcal{I}$ to satisfy $Tbox \wedge \{\neg A(a)\} \wedge \{C(a)\}$, which means A does not subsume C.

The binary clause containing C is relevant only to subsumption tests involving C. Since C is already positioned in the hierarchy— immediately subsumed by D, the binary clause and the concept C can be removed from the $Tbox$. The removal of such concepts and clauses may result in more concepts becoming negative, and possibly removable. For example, the removal of the clauses defining `man` and `woman`, makes `person` a negative

TOP $\equiv$ THING $\sqcup \neg$ THING

BOT $\equiv$ THING $\sqcap \neg$ THING

PERSON $\sqsubseteq$ HUMAN

MAN $\sqsubseteq$PERSON

WOMAN $\sqsubseteq$PERSON

```
{ top(X), thing(X) }
{ top(X), not(thing(X)) }
{ not(top(X)), thing(X), not(thing(X)) }
{ not(bot(X)), thing(X) }
{ not(bot(X)), not(thing(X)) }
{ bot(X), thing(X), not(thing(X)) }
{ not(person(X)), human(X) }
{ not(man(X)), person(X) }
{ not(woman(X)), person(X) }
```

Figure 6.5: Sample T-Box with Positive and Negative Concepts

concept present in only one clause. So that definition may be removed as well. It is also possible to remove positive concepts in this fashion. However, usually there are not many that satisfy the conditions for removal.

6.3.4 Results and Extensions

The number of subsumption comparisons that were actually made by the general-purpose system is shown in table 6.2. The number of possible subsumptions left after eliminating subsumptions is listed. The actual number of subsumption tests made were much less due to transitivity. We also note the performance of the prover when relevant clauses are used instead of the entire clause set. The total times taken to do all the subsumptions in each case are listed. The improvement in times obtained by using relevant clauses is crucial to obtaining reasonable classification speeds.

Figures 6.6 shows the performance of the classification technique on randomly generated knowledge bases. The number of subsumptions that remain to be tested after using *make_possible* on five random knowledge bases, is plotted against the number of concepts. Following Baader et al [BHN+92], the number of subsumption tests made relative

Classification after Preprocessing						
Knowledge Base	Total Concepts	Remaining Concepts	Possible Subsumptions	Subsumption Tests Made	Time taken (sec)	
					Relevant	All
CKB	80	48	157	7	19	> 500
FSS	132	64	9	2	11	> 300
Espresso	145	55	8	3	12	> 500
Companies	115	65	0	0	—	—

Table 6.2: Subsumption Tests performed by RRTP and Model Finder

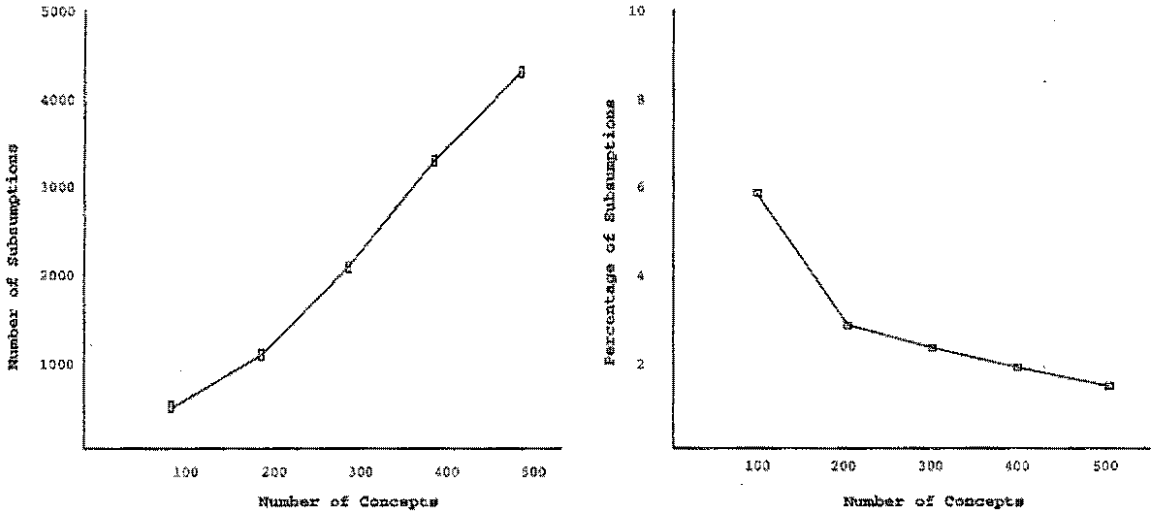

Figure 6.6: Plot on the left shows size of relation *possible*. Plot on the right shows size of *possible* as percentage of all possible subsumptions.

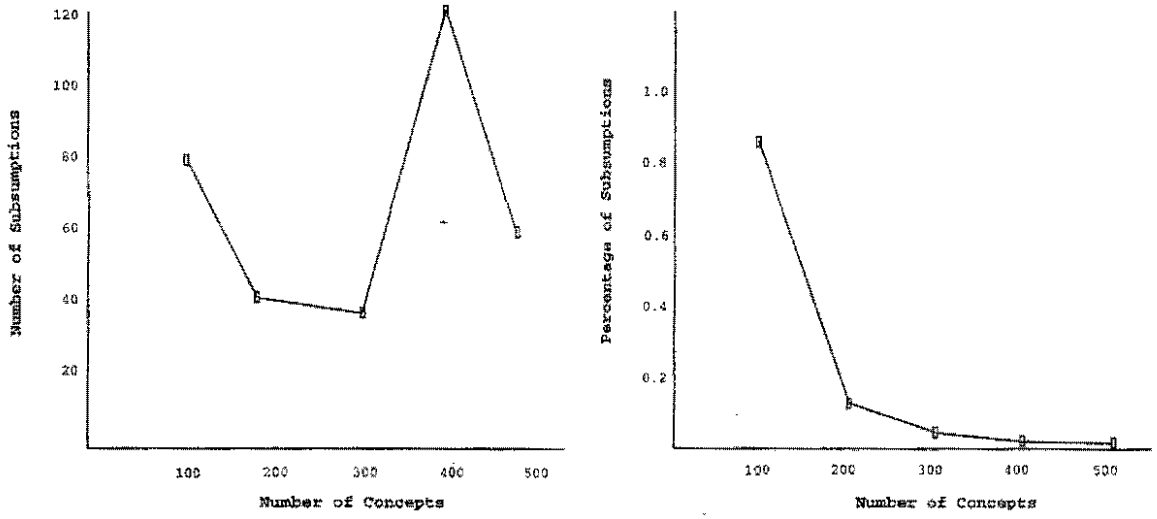

Figure 6.7: Classifying randomly generated knowledge bases. Plot on the left shows size of relation *possible/partial*. Plot on the right shows size as percentage of all possible subsumptions.

to the number of all the possible subsumptions tests is also plotted. In each case, the number of subsumption tests to be made is a very small percentage of the quadratic worst-case. Furthermore, several of the subsumptions are easily obtained by simply examining the clause set and determining implications and other trivial subsumptions. This further reduces the number of subsumption tests to be made as shown in Figure 6.7.

A good strategy to use if the knowledge-base is made of unrelated subsets is to separate the knowledge-base into these subsets. This can be done by first finding all the clauses in all the alternating paths from some clause. In this way all the unrelated subsets can be found. Introducing auxiliary concepts has been used in speeding up the classification process [Neb90]. This idea can be used here as well. We anticipate that such an extension would result in more obvious subsumptions detected and subsequently bring down classification time.

6.4 Summary and Conclusions

We believe that the contribution this paper makes is to show that state of the art theorem provers are quite efficient and practical. We do not claim that such general-purpose systems are capable of replacing description logic systems, but it is interesting that a general-purpose prover performs comparably with specialized techniques. Besides,

some description logic systems have improved since [HKNP92]. We note that some realistic knowledge-bases when translated run into several hundreds of clauses. We wonder how well this system would scale up to knowledge-bases that may result in tens of thousands, even hundreds of thousand clauses.

Although this paper has mostly discussed inferencing in the terminological component, we believe that it is easy to accommodate assertional reasoning using the same techniques. Assertions can be represented as unit clauses. However, asserting a large number of distinct individuals may make the model sizes too big.

A point to be made is that although the underlying logic for many description logic systems fall under the Goedel class of formulas, that is, the quantifier prefix is $\exists\forall\forall\exists$ [FLTZ93], the theorem prover and finite-model finder combination would work well for any extensions to concepts language provided the underlying logic still has the finite model property. Baader and Hollunder[BH91] state that KRIS was designed to include most of the description logic constructs with the restriction that the reasoner faces decidable problems when determining inferences such as subsumption checking. This is a very descriptive language; it also corresponds to some sublogic of first-order logic that has the finite model property [HN90]. In fact [HN90] point out that the semantic tableaux calculus used in KRIS combines the characteristics of a theorem prover and a finite model finder for that sublogic. However, KRIS is not complete for all sublogics of first-order logic that have the finite model property. Of course, it is not intended to be complete, but there may be features in the future that warrant inclusion in a description logic system and the tableaux calculus may require substantial revision to accommodate such changes. On the other hand the system we propose is already complete for all sublogics that have the finite-model property.

Buchheit et al[BDS93] introduce inclusion as a fundamental concept forming operator. For example, to express two concepts to be equivalent, it is sufficient to state that each one includes the other. Inclusion can easily be incorporated into the common terminological languages, and the the theorem prover and model-finder already provide a reasoning capability for it. On the other hand, some features are hard to represent in our system: A key feature of most description logics is number restriction as a concept-forming operator. Although this can be easily modeled in first-order logic for small values of the numbers, for large values the corresponding clauses become unmanageable. Such constructs cause problems for description logic systems as well.

We developed a number of preprocessing techniques to improve the efficiency of

subsumption, and also for pruning the number of subsumption tests to be made during classification. These techniques work well within any first-order logic framework that has the finite model property. Without these techniques, the running times of the theorem prover would be considerably longer, and will not be competitive. We believe that since theorem provers did reasonably well compared to specialized techniques in this domain, it is appropriate to examine whether theorem provers perform as well in other domains of interest.

Chapter 7

Conclusions

7.1 Conclusions

In this dissertation, we described the construction of a first-order theorem prover, a first-order finite model finder, and a proof procedure for Horn theories. The procedures were proved to be sound and complete for their respective tasks. All of these procedures were instance-based: the implementations generated instances. Our implementations relied on a good propositional decision procedure[ZS94]. We also used our prover and model-finder as a decision procedure in the area of Description Logics. We described performance results for all implementations.

We first described the theorem prover. RRTP, in the tradition of Herbrand procedures consists of two components: an instance generator and a propositional calculus prover. RRTP used the idea of "replacement" to generate instances. We formalized this notion of replacement and described the use of some combination of replacement and instantiation to create a complete theorem prover. UR-resolution is also added to RRTP to make it more powerful. RRTP performs respectably on theorems designated as "difficult" for state-of-the-art theorem provers. It is particularly effective on problems that do not involve equality.

The poor performance of RRTP on certain Horn problems prompted us to look for a suitable way to solve Horn problems. Our objective was to develop a strategy that would be goal sensitive and yet avoid the redundancy in proof search associated with such backward chaining techniques. We developed a procedure that creates instances that are sensitive to the goal but unlike classical backward chaining procedures our procedure never combines

the instances. Instead, we reason forward among the instances to arrive at the proof. We use UR-resolution to reason forward and obviate the need for using the propositional prover. We have added this procedure to RRTP to improve RRTP's performance on Horn problems.

Next, we described a simple first-order finite model finder. The model-finder is extremely useful in finding models for satisfiable formulas that have small models. In our observation, non-theorems obtained from theorems by making minor changes in the clauses, typically have very small models. As a result the model-finder is useful for detecting minor mistakes in theorems, which theorem provers usually do not detect and potentially run forever. The model-finder is based on a satisfiability preserving transformation of the first-order formula to a propositional formula.

Finally, we explored using our theorem-prover and model-finder as a decision procedure in Description Logics. The chief reasoning component provided by Description Logic systems is testing concept subsumption. Using tests devised to test Description Logics, we came to the conclusion that our combination performs compares well with the inferential abilities of description logic systems. We were only bested by systems that were limited in expressiveness. We also developed some techniques to make any prover run faster when doing subsumption testing. We also described some novel techniques for checking several subsumptions rapidly.

7.2 Extensions

We first note that there is enormous interest in the area of developing efficient solutions to the propositional satisfiability problem. Several real-world problems are being encoded as satisfiability problems, for example see [KS92],[Lar92] [CB94]. With areas such as Natural Language Processing and Machine Learning now benefiting from satisfiability methods, it is our belief that the continued research into this area will create better algorithms and more importantly better implementations of satisfiability methods. Naturally, such improvements stand to improve existing instance-based techniques, including ours.

We describe some possible extensions to our work.

Replacement Rule Theorem Prover

RRTP is very good with range-restricted clauses. The instance generation phase of RRTP can slow it down by creating too many instances that are not relevant to the

proof. We look to suppress the instantiation phase further by combining replacement with hyper-linking. RRTP can also be used to create replacement instances for other provers to use. We have observed that this has improved performance in at least one other prover.

RRTP has minimal equality support. This is evident from its performance in problems that require equality reasoning. There are some ways in which an efficient equality mechanism can be added to such an instance-based technique. One idea is to simulate paramodulation, an equality technique that combines well with resolution.

Proof Procedure for Horn Theories

The proof procedure for Horn theories is well suited for problems whose Horn clauses where the consequent literal is larger than all the antecedent literals. However, some work needs to be done to improve it for all Horn problems. We described how caching is useful to avoid repeating proofs. We believe that this idea will have an impact on all kinds of Horn problems. We also have some ideas on extending this approach to a complete theorem prover. A simple clause transformation of all clauses to Horn clauses in a manner resembling the prover SPRFN[Pla88b] may not be very efficient.

Description Logics

Description Logics provide several non-reasoning facilities to the user. These include editing tools and graphical depiction of concept hierarchies etc. We do no intend to provide such facilities. The entire point of our exercise was to test the strength of our system and offer some insight to future description logic system designers, and not replace description logic systems. It would also be interesting to see how the theorem prover and model-finder combination works as a decision procedure on other domains such as modal logics.

Appendix A

Prover Runs on Difficult TPTP Problems

The performance of the different provers on the various TPTP problems is detailed in this Appendix. For Otter and RRTP, since they were executed on the same machine with the same time bound (300s) the times taken for the proof is provided. All times are in seconds. For the other provers, only an indication of whether the proof was obtained is presented.

Table A.1: Comparison of Some Provers on difficult TPTP problems

Comparison of Some Provers on difficult TPTP problems						
Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
ANA002-2	N	N	N	N	N	N
BOO003-1	17.7	Y	Y	Y	Y	8.8
BOO004-1	2.8	Y	Y	N	Y	8.6
BOO005-1	10.0	Y	Y	N	Y	8.9
BOO006-1	25.2	Y	Y	N	Y	8.9
BOO007-1	245.1	N	N	N	N	N
BOO008-1	236.7	N	N	N	N	N
BOO009-1	26.3	N	N	N	Y	N
BOO010-1	11.6	N	N	N	Y	22.8
BOO012-1	76.8	Y	N	N	Y	N
BOO014-1	N	N	N	N	N	N
BOO015-1	N	N	N	N	N	N
BOO016-1	12.7	N	N	N	Y	27.0
BOO017-1	35.7	N	N	N	Y	N
CAT001-1	5.7	N	N	N	N	N
CAT001-3	1.9	Y	N	N	Y	98.2
CAT001-4	155.8	Y	Y	N	Y	24.2
CAT002-1	9.9	N	N	N	N	N
CAT002-3	145.9	Y	N	N	Y	4.2

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
CAT002-4	12.6	Y	Y	N	Y	6.4
CAT003-1	4.9	N	N	N	N	N
CAT003-2	N	Y	N	N	N	N
CAT003-3	N	Y	Y	N	Y	2.1
CAT003-4	134.9	Y	Y	N	Y	0.9
CAT004-1	9.6	N	N	N	N	N
CAT004-3	N	N	N	N	Y	77.7
CAT004-4	N	N	Y	N	Y	58.1
CAT005-3	0.4	N	N	N	Y	N
CAT005-4	0.6	N	Y	Y	Y	N
CAT006-3	0.3	N	N	N	Y	9.8
CAT006-4	0.4	N	Y	Y	Y	152.6
CAT008-1	2.6	N	N	Y	Y	18.6
CAT009-1	3.3	N	N	N	N	N
CAT009-3	N	N	N	N	N	N
CAT009-4	N	N	N	N	N	N
CAT010-1	3.5	N	N	N	N	N
CAT010-4	269	N	N	N	N	N
CAT011-3	144.6	N	N	N	Y	N
CAT011-4	7.7	N	Y	Y	Y	112.7
CAT012-3	65.7	Y	N	Y	Y	0.6
CAT013-3	143.7	Y	N	Y	Y	0.6
CAT014-3	149.1	N	N	N	Y	N
CAT014-4	6.6	N	Y	Y	Y	N
CAT016-3	0.2	Y	N	Y	Y	0.1
CAT017-3	0.1	Y	N	Y	Y	0.1
CAT018-1	2.7	Y	Y	Y	Y	143.7
CID001-1	0.1	N	N	N	N	N
CID003-2	258.8	N	N	N	N	N
COL002-2	10.4	Y	N	N	Y	N
COL002-3	N	Y	Y	Y	Y	115.3
COL003-3	N	N	N	N	N	N
COL003-4	N	N	N	N	N	N
COL003-5	N	N	N	N	N	N
COL003-6	N	N	N	N	N	N
COM003-1	N	N	N	N	N	N
GEO001-1	N	N	N	N	N	73.6
GEO001-2	N	N	Y	N	N	N
GEO002-1	237.1	N	N	Y	N	N
GEO002-2	N	N	N	Y	N	N
GEO003-1	1.1	Y	Y	Y	Y	2.3
GEO003-2	1.4	Y	Y	Y	Y	1.1
GEO006-1	N	N	N	N	N	N

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
GEO006-2	N	N	N	N	N	N
GEO010-1	N	N	N	N	N	N
GEO010-2	N	N	N	N	N	N
GEO011-1	N	N	N	N	N	N
GEO025-2	N	N	N	Y	N	N
GEO026-2	4.6	Y	Y	Y	Y	12.2
GEO027-2	N	N	Y	N	N	5.7
GEO030-2	34.9	N	N	Y	Y	80.5
GEO036-2	116.6	N	N	Y	Y	N
GEO037-2	N	N	N	N	N	N
GEO039-2	8.2	N	N	Y	Y	N
GEO040-2	N	N	N	N	N	86.9
GEO041-2	N	N	N	N	N	N
GEO042-2	N	N	N	N	N	N
GEO043-2	N	N	N	N	N	N
GEO048-2	N	N	N	N	N	N
GEO058-2	9.8	Y	N	Y	Y	N
GEO059-2	N	N	N	N	N	56.3
GEO064-2	N	N	N	N	N	N
GEO065-2	N	N	N	N	N	N
GEO066-2	N	N	N	N	N	N
GEO067-2	1.8	N	N	Y	N	N
GEO076-4	N	N	N	N	N	N
GEO077-4	N	Y	N	N	N	28.5
GRP008-1	11.0	Y	Y	Y	Y	16.2
GRP012-2	1.2	Y	Y	Y	Y	81.9
GRP012-3	4.2	Y	N	N	Y	17.4
GRP025-2	N	Y	Y	N	Y	N
GRP026-2	N	Y	Y	N	Y	N
GRP027-1	N	Y	Y	Y	Y	N
GRP029-1	0.2	Y	Y	Y	Y	N
GRP035-3	1.1	N	N	Y	Y	N
GRP037-3	1.2	Y	N	Y	Y	N
GRP039-1	N	N	N	N	N	N
GRP039-4	N	N	N	N	N	N
GRP040-3	N	N	N	N	N	N
GRP048-2	0.1	N	N	Y	Y	N
GRP051-1	N	N	N	N	N	N
GRP056-1	N	N	N	N	N	N
GRP057-1	N	N	N	N	N	N
GRP072-1	N	N	N	N	N	N
GRP074-1	N	N	N	N	N	N
GRP075-1	134.4	N	N	N	N	N

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
GRP076-1	268.7	N	N	N	N	N
GRP077-1	N	N	N	N	N	N
GRP078-1	202.4	N	N	N	N	N
GRP079-1	90.4	N	N	N	N	N
GRP080-1	N	N	N	N	N	N
GRP085-1	2.0	N	N	N	N	N
GRP086-1	1.8	N	N	N	N	N
GRP087-1	10.9	N	N	N	N	N
GRP097-1	N	N	N	N	N	N
GRP099-1	N	N	N	N	N	N
GRP100-1	151.8	N	N	N	N	N
GRP101-1	N	N	N	N	N	N
GRP102-1	N	N	N	N	N	N
GRP103-1	185.9	N	N	N	N	N
GRP105-1	N	N	N	N	N	N
GRP108-1	N	N	N	N	N	N
HEN003-1	0.8	N	N	Y	Y	89.8
HEN003-3	0.3	N	Y	Y	Y	2.5
HEN004-1	3.3	N	N	N	N	N
HEN005-1	1.1	N	N	N	Y	N
HEN006-1	3.8	N	N	N	N	N
HEN006-3	14.4	N	N	N	Y	N
HEN006-5	3.4	N	N	N	N	N
HEN007-1	18.1	N	N	N	N	N
HEN007-3	47.7	N	N	N	N	N
HEN007-5	17.0	N	N	N	N	N
HEN008-1	0.8	N	Y	Y	Y	5.4
HEN008-3	0.3	Y	N	Y	Y	4.4
HEN009-1	15.6	N	N	N	N	N
HEN009-3	119.8	N	N	N	N	N
HEN009-5	38.3	Y	Y	Y	Y	N
HEN010-1	12.2	N	N	N	N	N
HEN010-5	45.8	N	N	N	N	N
HEN011-1	110.2	N	N	N	N	N
HEN012-1	1.6	N	N	Y	Y	N
HEN012-3	0.3	N	Y	Y	Y	4.7
LAT005-5	36.6	N	N	N	N	N
LAT005-6	56.6	N	N	N	N	N
LCL146-1	37.9	N	N	N	N	N
LCL182-1	13.8	Y	Y	N	Y	213.7
LCL187-1	0.2	Y	Y	Y	Y	0.3
LCL192-1	7.0	Y	Y	Y	Y	1.8
LCL194-1	N	Y	Y	N	Y	1.2

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
NUM283-1	3.8	Y	N	N	N	247.6
NUM284-1	0.2	Y	Y	Y	N	81.4
PLA004-1	N	Y	N	N	N	26.5
PLA004-2	N	Y	N	N	N	26.3
PLA005-1	N	Y	N	N	N	25.4
PLA005-2	N	Y	N	N	N	25.2
PLA007-1	N	Y	N	N	N	5.4
PLA008-1	N	Y	N	N	N	N
PLA009-1	N	Y	N	N	N	5.2
PLA009-2	N	Y	N	N	N	5.2
PLA010-1	N	Y	N	N	N	N
PLA011-1	N	Y	N	N	N	27.6
PLA011-2	N	Y	N	N	N	27.3
PLA012-1	N	Y	N	N	N	N
PLA013-1	N	Y	N	N	N	26.7
PLA014-1	N	Y	N	N	N	26.6
PLA014-2	N	Y	N	N	N	26.5
PLA015-1	N	Y	N	N	N	N
PLA016-1	N	Y	N	N	N	11.5
PLA018-1	N	Y	N	N	N	N
PLA019-1	N	Y	N	N	N	11.1
PLA021-1	N	Y	N	N	N	15.5
PLA022-1	N	Y	N	Y	N	1.5
PLA022-2	N	Y	N	Y	N	1.5
PLA023-1	N	Y	N	N	N	N
RNG001-1	27.2	N	N	N	Y	N
RNG002-1	0.9	Y	Y	Y	Y	1.6
RNG003-1	0.7	Y	Y	Y	Y	1.5
RNG004-1	16.0	N	N	N	N	N
RNG005-1	3.8	N	N	Y	Y	97.3
RNG006-3	8.7	N	N	N	N	N
RNG007-1	12.3	N	N	N	N	N
RNG008-1	100.8	N	N	N	N	N
RNG037-1	3.8	N	N	Y	Y	97.8
RNG038-1	0.9	N	Y	N	Y	N
RNG039-1	11.2	N	N	N	N	N
RNG040-1	0.9	Y	Y	N	Y	0.8
RNG041-1	3.8	Y	Y	N	Y	N
ROB011-1	0.3	N	N	N	N	N
ROB016-1	0.4	N	N	N	Y	N
SET005-1	N	Y	N	N	Y	0.5
SET007-1	N	Y	N	N	Y	0.7
SET008-1	3.2	Y	Y	N	Y	0.3

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
SET009-1	1.9	Y	N	N	Y	0.5
SET011-1	257.1	Y	N	N	Y	0.3
SET012-1	N	N	N	N	N	0.5
SET012-2	N	N	N	N	N	6.2
SET013-1	N	N	N	N	N	0.6
SET013-2	N	N	N	N	N	7.3
SET014-2	N	Y	N	N	Y	4.0
SET015-1	N	N	N	N	N	0.6
SET015-2	N	N	N	N	N	7.2
SET017-6	65.2	N	N	N	N	223.1
SET019-4	10.1	N	N	N	N	N
SET024-4	1.4	Y	N	N	N	43.3
SET024-6	0.6	Y	Y	N	Y	2.6
SET025-4	0.8	Y	N	N	Y	2.4
SET025-6	1.4	Y	Y	N	Y	2.5
SET025-9	N	N	N	N	N	N
SET027-4	N	N	N	N	Y	N
SET027-6	3.2	N	N	N	Y	25.6
SET031-4	N	N	N	N	N	N
SET041-4	N	N	N	N	N	N
SET050-6	4.8	Y	N	N	Y	10.9
SET051-6	4.8	Y	N	N	Y	10.8
SET055-6	25.4	Y	N	N	N	33.7
SET061-6	59.3	N	N	N	N	N
SET062-6	18.9	N	N	N	Y	N
SET063-6	18.9	N	N	N	Y	N
SET064-6	101.9	N	N	N	Y	N
SET067-6	N	N	N	N	N	N
SET068-6	N	N	N	N	N	N
SET071-6	N	N	N	N	Y	N
SET072-6	170.8	N	N	N	N	224.7
SET073-6	58.5	N	N	N	N	N
SET074-6	56.8	N	N	N	N	N
SET075-6	47.7	N	N	N	N	N
SET076-6	N	N	N	N	Y	N
SET078-6	0.7	Y	Y	N	Y	8.4
SET079-6	65.2	N	N	N	N	N
SET080-6	N	N	N	N	Y	12.5
SET081-6	0.9	Y	N	N	Y	2.7
SET082-6	108.2	N	N	N	Y	N
SET083-6	108.5	N	N	N	Y	8.8
SET084-6	5.6	N	N	N	Y	9.0
SET085-6	126.1	N	N	N	Y	9.7

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
SET238-6	N	N	N	N	N	18.4
SET239-6	N	Y	N	N	Y	3.3
SET240-6	N	N	N	N	Y	12.3
SET241-6	N	N	N	N	Y	12.1
SET242-6	N	Y	Y	N	Y	3.4
SET243-6	N	N	N	N	N	N
SET245-6	N	N	N	N	N	N
SET252-6	N	N	N	N	Y	N
SET253-6	N	N	N	N	N	N
SET261-6	N	N	N	N	N	N
SET286-6	N	N	N	N	N	N
SET411-6	32.6	N	N	N	Y	11.7
SET454-6	N	N	N	N	N	N
SET479-6	149.6	Y	N	N	Y	24.9
SET506-6	24.4	N	N	N	N	271.5
SET507-6	39.1	N	N	N	N	N
SET510-6	124.3	N	N	N	N	N
SET516-6	N	N	N	N	N	N
SET517-6	N	N	N	N	N	N
SET553-6	N	N	N	N	Y	N
SET558-6	1.0	N	N	N	N	7.33
SET559-6	248.1	N	N	N	N	N
SET561-6	281.1	N	N	N	N	N
SET562-6	N	N	N	N	N	N
SET564-6	78.7	N	N	N	N	N
SET565-6	242.2	N	N	N	N	N
SET566-6	87.0	N	N	N	N	22.7
SYN014-1	N	N	N	N	N	N
SYN015-1	N	N	N	N	N	N
SYN113-1	1.3	Y	Y	Y	Y	5.2
SYN122-1	1.0	Y	Y	Y	Y	9.4
SYN137-1	1.1	Y	Y	Y	Y	10.9
SYN139-1	2.7	N	Y	Y	Y	56.5
SYN140-1	2.7	N	Y	Y	Y	56.4
SYN142-1	2.7	Y	Y	Y	Y	57.4
SYN143-1	2.8	N	Y	Y	Y	57.3
SYN155-1	1.6	N	Y	Y	Y	17.2
SYN157-1	1.6	Y	Y	Y	Y	9.6
SYN159-1	2.3	N	Y	Y	Y	22.4
SYN161-1	1.6	Y	Y	Y	Y	9.5
SYN163-1	2.2	N	Y	Y	Y	22.4
SYN176-1	1.0	Y	Y	Y	Y	6.7
SYN178-1	2.5	Y	Y	Y	Y	8.6

Problem	Otter	SETHEO	CLIN	CLIN-E	Linus	RRTP
SYN179-1	1.5	Y	Y	Y	Y	12.6
SYN181-1	1.9	Y	Y	Y	Y	10.3
SYN182-1	1.6	Y	Y	Y	Y	7.9
SYN190-1	1.5	Y	Y	Y	Y	13.1
SYN192-1	1.5	Y	Y	Y	Y	9.2
SYN198-1	0.9	Y	Y	Y	Y	3.0
SYN200-1	0.8	Y	Y	Y	Y	3.1
SYN205-1	2.6	Y	Y	Y	Y	56.8
SYN206-1	1.8	Y	Y	Y	Y	8.1
SYN207-1	1.6	Y	Y	Y	Y	45.5
SYN218-1	0.9	Y	Y	Y	Y	3.0
SYN219-1	1.0	Y	Y	Y	Y	6.3
SYN235-1	2.0	Y	Y	Y	Y	43.2
SYN252-1	2.8	N	Y	Y	Y	56.3
SYN253-1	2.7	N	Y	Y	Y	56.7
SYN254-1	2.7	N	Y	Y	Y	56.7
SYN263-1	1.9	Y	Y	Y	Y	4.0
SYN271-1	2.4	Y	Y	Y	Y	18.6
SYN272-1	2.2	Y	Y	Y	Y	6.5
SYN298-1	1.8	Y	Y	Y	Y	8.1
SYN300-1	1.9	Y	Y	Y	Y	8.1

Appendix B

Description Logics

Table B.1 tabulates the performances of the different systems on "hard" examples. The examples were designed to bring out bad performances in description logic systems. The description logic system times for these tests are taken from [HKNP92]. These times are not comparable at all as they were run on different hardware. Some of these times may not be appropriate as some of these systems have been improved upon. For a more accurate comparison, we performed the tests on two systems, FLEX and BACK, that were available to us, on the platform that the theorem prover and model finder ran on. These results are shown in Table B.2. The time given for FLEX is misleading because the system classifies the knowledge base upon input. The times provided for FLEX are the classification times.

Hard Inferences							
Result of Test (sec)	System						
	BACK	CLASSIC	KRIS	LOOM	MESON	SBONE	RRTP*
1 (a)	1	2	3	1	1	11	1
1 (b)	1	4	77	3	2	33	3
1 (c)	2	5	2680	5	6	56	8
2 (a)	7	3	82	7	3	99	3
2 (b)	32	11	1867	22	23	859	16
2 (c)	75	16	—	39	84	3263	45
3 (a)	25	4	459	28	29	372	7
3 (b)	352	40	18230	155	5099	1836	15
3 (c)	6035	706	—	666	—	9500	30
4 (a)	—	—	4	4†	—	—	84
4 (b)	—	—	49	8†	—	—	>10000§
4 (c)	—	—	745	13†	—	—	—§

* With finite-model finder

† Inference not computed.

§ When inferences were aided by lemmas times dropped to 200s and 800s respectively.

Table B.1: Hard Cases on Various Platforms

Hard Inferences			
Result of Test (sec)	System		
	BACK	FLEX	RRTP*
1 (a)	1	6	1
1 (b)	1	7	3
1 (c)	2	13	8
2 (a)	6	27	3
2 (b)	14	138	16
2 (c)	45	446	45
3 (a)	22	25	7
3 (b)	252	59	15
3 (c)	3114	118	30
4 (a)	—	12	84
4 (b)	—	49	
4 (c)	—	186	

Table B.2: Hard Cases on the DEC 5000/120

Bibliography

[Ale95] Geoffrey Alexander. *Proving first-order equality theorems with hyper-linking.* PhD thesis, University of North Carolina at Chapel Hill, 1995.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 243–320. Elsevier Science Publishers, Amsterdam, 1990.

[BC74] George S. Boolos and Richard C.Jeffrey. *Computability and Logic.* Cambridge University Press, Cambridge, 1974.

[BDS93] Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. Artificial Intelligence Research*, 1:109–138, 1993.

[BH91] F. Baader and B. Hollunder. KRIS: Knowledge Representation and Inference System. *SIGART Bulletin*, 2:22–27, June 1991.

[BHN+92] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jurgen Profitlich, and Enrico Franconi. An empirical analysis of optimization techniques for terminological representation systems. In *Principles of Knowledge Representation and Reasoning— Proceedings of the 3rd International Conference*, 1992.

[BPS94] Alex Borgida and Peter F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic system. *J. Artificial Intelligence Research*, 1:277–308, 1994.

[Bra92] Julian C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhauser, Moston, Mass., 1992.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knoweldge representation system. *Cognitive Science*, 9:171–216, April 1985.

[CA93] J.M. Crawford and L.D Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 21–27, 1993.

[CB94] J.M. Crawford and A.B Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

Bibliography

[Ale95] Geoffrey Alexander. *Proving first-order equality theorems with hyper-linking.* PhD thesis, University of North Carolina at Chapel Hill, 1995.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 243–320. Elsevier Science Publishers, Amsterdam, 1990.

[BC74] George S. Boolos and Richard C.Jeffrey. *Computability and Logic.* Cambridge University Press, Cambridge, 1974.

[BDS93] Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. Artificial Intelligence Research*, 1:109–138, 1993.

[BH91] F. Baader and B. Hollunder. KRIS: Knowledge Representation and Inference System. *SIGART Bulletin*, 2:22–27, June 1991.

[BHN+92] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jurgen Profitlich, and Enrico Franconi. An empirical analysis of optimization techniques for terminological representation systems. In *Principles of Knowledge Representation and Reasoning— Proceedings of the 3rd International Conference*, 1992.

[BPS94] Alex Borgida and Peter F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic system. *J. Artificial Intelligence Research*, 1:277–308, 1994.

[Bra92] Julian C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhauser, Moston, Mass., 1992.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knoweldge representation system. *Cognitive Science*, 9:171–216, April 1985.

[CA93] J.M. Crawford and L.D Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 21–27, 1993.

[CB94] J.M. Crawford and A.B Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[CGT90] S. Ceri, G. Gottlob, and L .Tanca. *Logic Programming and Databases.* Springer Verlag, Berlin, 1990.

[Chu56] A. Church. *Introduction to Mathematical Logic Vol 1.* Princeton Univ. Press, Princeton, New Jersey, 1956.

[Chu94a] Heng Chu. *CLIN-S User's Manual*, 1994.

[Chu94b] Heng Chu. *Semantically Guided First-Order Theorem Proving Using Hyper-Linking.* PhD thesis, University of North Carolina at Chapel Hill, 1994.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[CM81] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[CP92] Heng Chu and David Plaisted. Generating unit consequences of a ground clause set. Technical report, University of North Carolina at Chapel Hill, 1992.

[DG79] Burton Dreben and Warren. D Goldfarb. *The Decision Problem: Solvable classes of Quantificational formulas.* Addison Wesley, Reading, Mass., 1979.

[Fit90] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, 1990.

[FL94] Christian Fermuller and Alexander Leitsch. Automated model building by hyperresolution. In *Automated Model Building Workshop CADE-12. Nancy, France*, pages 18–21, 1994.

[FLTZ93] C. Fermuller, A. Leitsch, T. Tammet, and N Zamov. *Resolution Methods and the Decision Problem.* Springer-Verlag, 1993. Lecture Notes in Artificial Intelligence, 679.

[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.

[Gog96] Joseph Goguen. *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, Mass., 1996.

[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.

[HKNP92] J. Heinsohn, D. Kudenko, B. Nebel, and H. Profitlich. An emperical analysis of terminological representation systems. Technical report, DFKI Research Report, German Research Center for Artificial Intelligence (DFKI), Kaiserlautern, 1992.

[HN90] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Technical report, DFKI Research Report RR-90-04, German Research Center for Artificial Intelligence (DFKI), Kaiserlautern, 1990.

[Jac89] Peter Jackson. *Logic-based Knowledge Representation.* MIT Press, Cambridge, Mass., 1989.

[Kol96] Gina Kolata. Computer math proof shows reasoning power. New York Times, December 10 1996.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings ECAI-92*, pages 359–363, 1992.

[KZ94] Sun Kim and Hantao Zhang. Modgen: Theorem proving by model generation. In *Proceedings AAAI-94*, 1994.

[Lar92] T. Larrabee. Efficient generation of test patterns using boolean satisfiability. *IEEE Transactions on CAD*, 11:4–15, 1992.

[Lee90a] Shie-Jue Lee. *CLIN: An Automated Reasoning System Using Clause Linking.* PhD thesis, University of North Carolina at Chapel Hill, 1990.

[Lee90b] Shie-Jue Lee. *CLIN: An Automated Reasoning System Using Clause Linking.* PhD thesis, University of North Carolina at Chapel Hill, 1990.

[Let97] R. Letz. LINUS: A Clause Linking Theorem Prover. *Journal of Automated Reasoning*, 18(2):205–210, 1997.

[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.

[LP94] Shie-Jue Lee and David. A. Plaisted. Use of replace rules in theorem proving. *Methods of Logic in Computer Science*, 1:217–240, 1994.

[LSBB92] R. Letz, J. Schumman, W. Beyerl, and W. Bibel. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[Mac91] R. MacGregor. Inside the LOOM description classifier. *SIGART Bulletin*, 2:88–82, June 1991.

[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.

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.

[Neb90] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems.* Springer-Verlag, Berlin, Germany, 1990.

[NS93] Anil Nerode and Richard A. Shore. *Logic for Applications.* Springer-Verlag, New York, 1993.

[Pet91] C. Petalson. The BACK system — an overview. *SIGART Bulletin*, 2:114–119, June 1991.

[Pla76] David.A Plaisted. *Theorem Proving and Semantic Trees.* PhD thesis, Stanford University, 1976.

[Pla82] David A. Plaisted. A simplified problem reduction format. *Artificial Intelligence*, 18:227–261, 1982.

[Pla84] David A. Plaisted. The occur-check problem in prolog. *New Generation Computing*, 2:309–322, 1984.

[Pla88a] David A. Plaisted. Non-Horn clause logic programming without contrapositives. *J. Automated Reasoning*, 4:287–325, 1988.

[Pla88b] David A. Plaisted. Non-Horn clause logic programming without contrapositives. *J. Automated Reasoning*, 4:287–325, 1988.

[Pla94] David A. Plaisted. The search efficiency of theorem proving strategies: An analytical comparison. Technical report, MPI-I-94-233, MPI Informatik, Saarbrucken, Germany, 1994.

[PP91] David A. Plaisted and Richard C. Potter. Term rewriting: Some experimental results. *J. Symbolic Computation*, 11:149–180, 1991.

[PP95] M. Paramasivam and David A. Plaisted. Automated deduction techniques for subsumption in concept languages. In *Proceedings of the International Conference on Intelligent Sytems,1995*, 1995.

[PP97a] M. Paramasivam and David A. Plaisted. A Replacement Rule Theorem Prover. *Journal of Automated Reasoning*, 18(2):221–226, 1997.

[PP97b] M Paramasivam and David A. Plaisted. Automated deduction techniques for classification in description logics. *J. Automated Reasoning*, Forthcoming, 1997.

[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.

[PSMB+91] P. F. Patel-Schneider, D. L. McGuiness, R. J. Brachman, L. Alperin Resnick, and A. Borgida. The CLASSIC knowledge representation system: Guiding principles and implementational rational. *SIGART Bulletin*, 2:108–113, June 1991.

[PZ97] David A. Plaisted and Yunshan Zhu. Ordered semantic hyper-linking. In *Proceedings AAAI-97*, 1997.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[SKM96] Bart Selman, Henry Kautz, and David McAllester. Computational challenges in propositional reasoning and search. Technical report, AT & T Laboratories, Tracking No.:A 828, 1996.

[Sla94] John Slaney. *FINDER, Finite Domain Enumerator: Version 3.0 Notes and Guide*, 1994. Automated Reasoning Project, Australian National University.

[SS97] C.B. Suttner and G. Sutcliffe. The Design of the CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2):139–162, 1997.

[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`.

[Tam90] Tanel Tammet. *Resolution Methods for Decision Problems and Finite-Model Building*. PhD thesis, Chalmers University of Technolgy, Goteborg, Sweden, 1990.

[Wal93] Adrian Walker. Backchain iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *J. Automated Reasoning*, 11:1–22, 1993.

[Zhu97] Yunshan Zhu. *Efficient Proof Procedures for First-Order Theorem Proving*. PhD thesis, University of North Carolina at Chapel Hill, 1997.

[ZS94] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, Department of Computer Science, University of Iowa, 1994.