

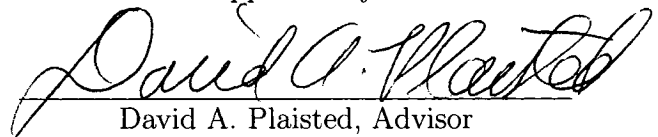
OSHL-U: A First Order Theorem Prover Using Propositional Techniques and Semantics

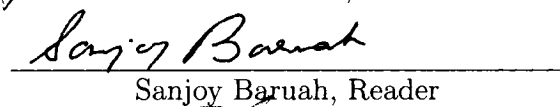
by
Swaha D. Miller

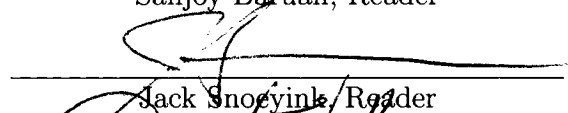
A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the department of Computer Science.

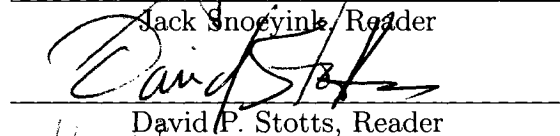
Chapel Hill
2005

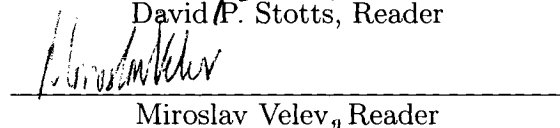
Approved by:

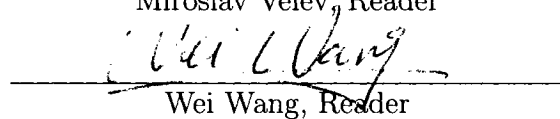

David A. Plaisted, Advisor


Sanjoy Baruah, Reader


Jack Snoeyink, Reader


David P. Stotts, Reader


Miroslav Velev, Reader


Wei Wang, Reader

UMI Number: 3200820

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3200820

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2005
Swaha D. Miller
ALL RIGHTS RESERVED

ABSTRACT

SWAHA D. MILLER: OSHL-U: A First Order Theorem Prover Using Propositional Techniques and Semantics. (Under the direction of David A. Plaisted.)

Automated theorem proving is the proving of theorems with an automatic computer program. Automation of theorem proving is useful in proving repetitive theorems quickly and accurately, as well as in proving theorems that may be too large or complex for humans to handle. Automated theorem proving finds application in various fields, such as, the verification of integrated circuits, software verification, deductive databases, mathematics, and education.

The early success of automated theorem provers for first-order logic based on the resolution-unification paradigm has caused theorem proving research to be directed largely towards resolution and its variants. The problems people looked at in the early days of automated deduction were mostly in the classes of problems that resolution is especially efficient at solving such as *Horn* problems (Every clause in such problems contains at most one positive literal) and *UR resolvable* problems (Problems in which the proof can be obtained purely by UR resolution, which is a restricted form of resolution). The initially good performance of resolution on such problems led to disillusionment later on. Problems that are hard to solve for humans are less and less likely to be Horn or UR resolvable, so resolution is less likely to be efficient on such problems. That few hard problems have been solved automatically by current provers further supports this. Therefore, there is a need to go beyond resolution for harder problems. Approaches based on propositional techniques applied to first-order theorem proving have great potential in this direction. Provers based on propositional techniques such as DPLL are used extensively for hardware verification but resolution is hardly ever used for solution of large propositional problems because it is far less efficient. Resolution has serious inefficiencies on non-Horn propositional problems and it is likely that these inefficiencies carry over also to first-order problems. In recent times, techniques developed for deciding propositional satisfiability perform at tremendous speeds, solving verification problems containing tens of thousands of variables and hard random 3SAT problems containing millions of variables. The desire to import such propositional efficiency into first-order theorem proving has revived an interest in propositional techniques for proving first-order logic problems.

This dissertation explores propositional techniques to first-order theorem proving and how these compare in performance to resolution-unification techniques. It formulates some tech-

niques that are shown to enhance the performance of OSHL, a theorem prover for first-order logic based on propositional techniques. The resulting implementation, OSHL-U, performs better than resolution on categories of problems that are hard for resolution. The performance of OSHL-U can further be improved by the use of semantics that provide problem-specific information to help the proof search; this is demonstrated experimentally in the dissertation. The techniques applied to enhance OSHL performance are applicable to other theorem provers, too, and contribute towards building more powerful theorem provers for first-order logic in general.

ACKNOWLEDGMENTS

I thank my advisor, David Plaisted, for introducing me to an exciting field of research and giving me the opportunity to work with him, and for his constant guidance, support and enthusiasm with our work. I also thank all the members of my committee for their valuable guidance and providing me with a broader perspective and constructive feedback throughout.

I thank the faculty of the Computer Science Department at UNC-Chapel Hill for the many ways in which they have taught and motivated me, both in and out of the classroom. I also thank the Professors at the Graduate School and Dr. and Mrs. Smithwick, who have given me valuable advice and generous support during the last year.

I am really grateful to my husband Dorian for always being there for me, brightening my days, and helping and inspiring me in innumerable ways - big and small - to complete this dissertation. I will always be indebted to my parents Shipra and Dipak, and my brother Satyaki, for their love and encouragement, and for being supportive of me at all those times when I needed them most. I have depended on these four people for my emotional well-being and without them in my life, this dissertation would not have been possible.

Last, but not the least, I thank the staff and students in the department, with whom I have constantly interacted for work and for play, and who have helped make my graduate school experience memorable.

TABLE OF CONTENTS

LIST OF TABLES	xv
LIST OF FIGURES	xvii
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Statement	5
1.3 Summary of Results	5
1.4 Outline of Thesis	7
2 Background on First-Order Automated Theorem Proving	8
2.1 General Terminology and Definitions	9
2.2 Problem Description	11
2.3 Conjunctive Normal Form for Representing First-Order Logic Formulae	11
2.4 Herbrand Sets and Herbrand's Theorem	14
2.5 Propositional Approach to First-Order Theorem Proving	14
2.5.1 Davis-Putnam-Logemann-Loveland (DPLL) Procedure	15
2.6 The Resolution Approach	16
2.6.1 Rule of Resolution	16
2.6.2 Unification	16

2.6.3	Resolution Theorem Proving	19
2.6.4	Further Definitions	20
2.7	Theorem Proving in First-Order Logic	20
2.7.1	The First-order DPLL Procedure	21
2.7.2	The Model Evolution Calculus	22
2.7.3	The Disconnection Theorem Prover	22
2.7.4	Propositional Techniques at UNC	23
2.8	The TPTP Problem Set	23
2.9	Strategy Selection and Performance Tuning in Modern Provers	24
2.9.1	Choice of a Resolution Prover for Experimental Comparison	25
3	Ordered Semantic Hyper-Linking with Unit Rules	26
3.1	The OSHL Strategy	27
3.2	OSHL extended with Unit Rules of Inference	28
3.2.1	Basic Rules in OSHL	30
3.2.2	U Rules	31
3.2.3	Order of Applying Rules in OSHL-U	33
3.3	Examples of OSHL-U Operation	35
3.4	Heuristics for Proof Search	42
3.4.1	Delta Size Measure for Clause Instances	42
3.4.2	Favoring Ground Terms in Generating Minimal Instances	43
3.4.3	Relevance Distance from Input Clauses	43
3.5	Implementation	44
3.5.1	Disunification	44
3.5.2	Eligibility Substitution	46

3.5.3	The Instantiation Algorithm	46
3.5.4	How Semantics Are Used	48
3.5.5	Proofchecker	49
4	Experimental Evaluation of OSHL-U Efficiency	51
4.1	Execution Time as Measure of Efficiency	51
4.2	Experimental Results	52
4.2.1	Execution Time	52
4.2.2	Performance Improvement from Unit Rules	56
4.2.3	OSHL-U Performance Compared to Other OSHL Enhancements . . .	56
4.3	Conclusions	58
5	Refinements to the OSHL-U Implementation	59
5.1	Effective Refinements to OSHL-U Implementation	59
5.1.1	Avoid Repeating Computations in Unit Filter and UR	59
5.1.2	Pruning Invalid Paths from the Search	60
5.1.3	Order of Input Clauses	60
5.1.4	Potential Unification	60
5.1.5	Potential Unification With Counting of Unifiers	61
5.1.6	Incremental Size Bound on U Rules	61
5.1.7	Instantiating Negative Literals Before Positive Literals	61
5.2	Refinements That Were Discarded	62
5.2.1	Unit Filtering on Units Generated by Non-ground UR Resolution . . .	62
5.3	Results	62
5.4	Conclusion	63

6	Space Efficiency of OSHL-U Strategy and Resolution	67
6.1	Search Space Efficiency	67
6.2	Importance of Storage Space for Theorem Prover Efficiency	67
6.3	Space Efficiency Comparison of OSHL-U and Otter	67
6.4	Conclusion	72
7	Comparison of Theorem Provers with Different Inference Rates	74
7.1	Motivation	75
7.2	Comparing Theorem Provers with Different Inference Rates	76
7.2.1	Time Efficiency	76
7.2.2	Space Efficiency	77
7.3	Relative Efficiency of OSHL-U and Otter	77
7.4	Time Efficiency	78
7.4.1	Comparison over all, Horn and non-Horn Problems	78
7.4.2	Performance on Field Theory and Group Theory Categories	80
7.5	Space Efficiency	80
7.6	Discussion	81
7.7	Drawbacks of the Comparison Techniques	84
7.8	Conclusion	86
8	Performance of OSHL-U on Problems Requiring Definition Expansion	87
8.1	Introduction	87
8.2	Survey of Approaches to Definition Expansion	88
8.3	Experiments	88
8.4	Discussion	96

8.5	Conclusions	97
9	Semantic Guidance in Proof Search	98
9.1	Use of Semantics in OSHL-U	100
9.1.1	An Example: “Who Killed Aunt Agatha?”	101
9.1.2	Semantics in Group Theory Problems	103
10	Conclusion	106
10.1	Future Work	109
	BIBLIOGRAPHY	111

LIST OF TABLES

1.1	Number of proofs obtained by Otter in various categories	3
1.2	Comparison of Otter and Hyper Linking on non-Horn problems	4
1.3	Comparison of OSHL using replacement rules and Otter	5
4.1	Proofs on Otter and OSHL-U compared by TPTP ratings	53
4.2	Proofs on Otter and OSHL-U compared over Horn category	54
4.3	Proofs on OSHL-U with and without U rules	57
4.4	Proofs on OSHL-U and OSHL with different methods	57
5.1	Proofs on Otter and OSHL-U compared by TPTP ratings	64
5.2	Proofs on Otter and OSHL-U compared over Horn category	65
5.3	Increase in number of proofs due to OSHL-U refinements	66
6.1	Distribution of ratio of clauses generated by Otter to OSHL-U	69
6.2	Number of clauses generated by Otter and OSHL-U and their ratio	69
6.3	Ratio of total search space of Otter to OSHL-U	70
6.4	Distribution of ratio of clauses stored by Otter to OSHL-U	72
6.5	Number of clauses stored by Otter and OSHL-U and their ratio	72
6.6	Ratio of total storage space of Otter to OSHL-U	73
8.1	Timing and clauses of different provers on p1 left associative theorems	91
8.2	Timing and clauses of different provers on p1 right associative theorems . . .	91
8.3	Timing and clauses of different provers on p2 left associative theorems	92
8.4	Timing and clauses of different provers on p2 right associative theorems . . .	92
8.5	Timing and clauses of different provers on p3 theorems	93
8.6	Timing and clauses of different provers on p4 left associative theorems	93

8.7	Timing and clauses of different provers on p4 right associative theorems . . .	94
8.8	Timing and clauses of different provers on p5 left associative theorems	94
8.9	Timing and clauses of different provers on p5 right associative theorems . . .	95
8.10	Timing and clauses of different provers on p6 theorems	95
9.1	Timing and clauses on Otter and OSHL-U with natural semantics	104
9.2	Timing and clauses on Otter and OSHL-U with non-natural semantics	104

LIST OF FIGURES

4.1	Execution times with OSHL-U and Otter	55
5.1	Execution times with the refined OSHL-U and with Otter	63
6.1	Scatter plot of number of clauses generated by OSHL-U and Otter	68
6.2	Scatter plot of number of clauses stored by OSHL-U and Otter	71
7.1	Time efficiency functions for Otter and OSHL-U on all problems	78
7.2	Time efficiency functions for Otter and OSHL-U on Horn problems	79
7.3	Time efficiency functions for Otter and OSHL-U on non-Horn problems	80
7.4	Time efficiency function for Otter over a long range	81
7.5	Time efficiency functions for Otter and OSHL-U on FLD problems	82
7.6	Time efficiency functions for Otter and OSHL-U on GRP problems	83
7.7	Space efficiency of Otter and OSHL-U over all problems	84
7.8	Space efficiency of Otter and OSHL-U over Horn problems	85
7.9	Space efficiency of Otter and OSHL-U over non-Horn problems	86

Chapter 1

Introduction

Automated theorem proving is the proving of mathematical theorems by a fully automatic computer program. Humans, of course, have been proving theorems without the aid of computers for thousands of years. But using a computer helps to prove theorems quickly and accurately. It may be easy to see the advantage of mechanizing theorem proving for large proofs involving computational effort of a repetitive nature. On such problems, humans tend to be more prone to mistakes, finding the work tedious and boring; use of computers helps to avoid such mistakes and also gives us the advantage of greater computational speed in proving these theorems. Mechanized theorem provers can also be advantageous on proofs that are difficult for humans because of the size of the problem, or its complexity. A notable example is the Robbins problem which used to be an open problem prior to 1996 when it was solved by the automated theorem prover EQP [McC97]. However, such cases are rare and exceptional; few hard problems have been solved automatically by current provers.

Automated theorem provers have been successfully used by researchers to tackle open questions in mathematics and logic, and to solve problems in engineering. Well-known commercial uses of automated theorem proving include design and verification of integrated circuits [DM96, KM96], software generation [Smi90, SWL⁺94], and software verification. Automated theorem proving also finds application in education [SM84, LBM98, MRS01], expert systems, deductive databases [SK88], planning and robotics. Potential uses of automated theorem proving could also be in semantic web and query answering [Tam03]. Since the Pentium FDIV bug, which cost Intel 475 million dollars, the complicated floating point units of modern microprocessors have been designed with extra scrutiny. In the latest processors from AMD, Intel, and others, automated theorem proving has been used to verify that the divide and other operations are correct [AJK⁺00, CB98, CCH⁺96, SR95, DA95].

However, automated theorem provers are still limited in capability. To date, it has not been possible to successfully automate much of the theorem proving ability of humans, especially

that involving expert knowledge. Humans often reason by constructing models of the situation and looking for counter-examples. Humans are able to draw upon a vast amount of problem-specific knowledge called the semantics of the problem, general problem-solving tactics, and prior experience with proving other theorems, when attempting to prove a theorem. Some of the aspects of human reasoning [SY97, MS98], such as “intuition” and “insight,” are not even precisely understood. It is often difficult to identify the relevant information that helps in proving theorems, and efficient ways of providing this information to a mechanized theorem prover have not been fully devised. Current uses of automated theorem provers still involve a great deal of human interaction and guidance.

The language in which a problem is presented to and manipulated by an automated theorem prover is called a logic. A logic allows a precise formal statement of the necessary information, which can then be manipulated by a theorem proving system. Unlike using a natural language such as English, there is no ambiguity in the statement of the problem, making logic a good formalism for automated reasoning. First-order logic can be used to represent various kinds of problems, such as in artificial intelligence, mathematical theories, and some network protocols [AB03, Bla03], to name a few. There are also hardware verification techniques which make use of translations of other formalisms to first-order logic [CHM02, VB03].

1.1 Motivation

The earliest attempts in automated reasoning were directed at solving problems in mathematical reasoning, which is a unique field in that it combines creativity and objectivity. There were two early approaches to automated reasoning. One was to try to understand and imitate the actions of mathematicians finding proofs. The other was to develop algorithms based on systematic logical reasoning. For a more detailed treatment of these two methods and a survey of the early history in automated reasoning, see [Dav83]. One of the first automated theorem provers, the Logic Theorist, developed by Newell, Shaw, and Simon in 1957 used human problem-solving techniques to solve reasoning problems in logic [NSS56, NSS63, NSS83]. Subsequently, better results in proving theorems were obtained by using tools of mathematical logic rather than being restrained to the imitation of humans, so the focus of the field shifted away from techniques based on human reasoning. Most present-day mechanized theorem provers for first-order logic are based on a strategy called resolution-unification [Rob65], which will be formally defined and summarized in Section 2.5 of this thesis. Resolution-unification consists of two techniques — resolution and unification. Resolution involves deriving new facts from existing facts. Unification involves matching facts to make them the same. Unification in first-order logic is a more difficult problem and

	non-Horn Problems			UR Resolvable Problems		
	UR res.	non UR res.	total	Horn	non-Horn UR res.	total
# problems	857	1915	2772	1644	857	2501
Otter proofs	400	533	933	764	400	1164
%proved	46.7	27.8	33.7	46.5	46.7	46.5

Table 1.1: Number of proofs obtained by resolution prover Otter in various categories. “UR res.” denotes UR resolvable problems. These statistics suggest that the power of resolution is mostly from problems that are Horn or UR resolvable.

requires more sophisticated algorithms than unification in propositional logic; unification in first-order logic is called true unification. Resolution-unification, often referred to simply as resolution, was instrumental in building one of the first viable automated theorem provers. Since this early success of resolution, theorem proving research has largely been directed towards resolution and its variants.

Resolution obtained its initial success because the easy problems people looked at in the early days of automated deduction fell mostly in the classes of problems that resolution is especially good at solving, such as Horn problems (problems in which every clause contains at most one positive literal) and UR resolvable problems (problems in which the proof can be obtained purely by UR resolution, which is a restricted form of resolution). The good performance of resolution on such problems gave it an advantage but led to disillusionment later on. As the problems become harder, it is less and less likely that they would be Horn or UR resolvable, so resolution is less likely to be efficient on such problems. Hence there is a need to go beyond resolution for hard problems.

We ran some tests with a well-known resolution prover, Otter [McC03], on problems from the TPTP problem set [SS98a], in order to see how the performance varies with the Horn and UR resolvable property of theorems. UR resolvability is an undecidable property for first-order logic problems. Because all the test problems are unsatisfiable, and every unsatisfiable problem that is Horn is also UR resolvable, we can classify the Horn problems as UR resolvable. For the non-Horn problems, we approximate UR resolvability by running each problem with only UR resolution strategy for upto 5 minutes; if a proof is found, then we can conclude that the problem is UR resolvable. The results are summarized in Table 1.1. Otter obtains proofs for at least 46.5 per cent of the UR resolvable problems and at most 27.8 per cent of the non-UR resolvable problems. This indicates that resolution gets much of its strength from UR resolvable problems.

Humans use a variety of problem-solving techniques such as representing information in diagrams, looking for examples and counter-examples, analysing a problem on a case-by-

Name of Problem	Number of Input Clauses	Otter Time	Hyper Linking Time
Ph5	45	38606.76 seconds	1.8 seconds
Ph9	297	>24 hours	2266.6 seconds
Latinsq	16	>24 hours	56.4 seconds
Salt	44	1523.82 seconds	28.0 seconds
Zebra	128	>24 hours	866.2 seconds

Table 1.2: Comparison of resolution strategy in Otter and Hyper Linking strategy on non-Horn problems quoted from [PL92].

case basis, and using the meanings of symbols in a problem. Methods have been proposed to incorporate human reasoning techniques into mechanical reasoning through the use of semantic models and diagrams to guide theorem provers. Notable successes in that regard are the Geometry Theorem Prover [GHL63] and more recently, automated theorem provers using the area method [CGZ96, CG01]. Finding effective ways of representing and using semantics in automated theorem provers continues to be a challenging problem.

Automated theorem proving research has largely been directed towards provers based on the resolution-unification paradigm; other approaches to theorem proving have been explored relatively little. One such approach is based on propositional techniques applied to first-order theorem proving. In recent times, techniques developed for deciding propositional satisfiability perform at tremendous speeds, solving verification problems containing hundreds of thousands of variables and hard random 3SAT problems containing millions of variables [Vel04b, Vel04a]. The desire to import such propositional efficiency into first-order theorem proving has revived an interest in propositional techniques for proving first-order logic problems.

Provers based on propositional techniques such as DPLL [DLL62] are used extensively for hardware verification but resolution is hardly ever used for solution of large propositional problems because it is far less efficient. Resolution has serious inefficiencies on non-Horn propositional problems and it is likely that these inefficiencies carry over also to first-order problems. It has been shown that DPLL style techniques are more efficient than resolution on non-Horn propositional problems [PL92]. Table 1.2 quotes results from [PL92], which show that Hyper Linking, a propositional style technique similar to DPLL, outperformed the most well-known resolution prover at that time, Otter, on some first-order problems. Table 1.3 quotes the results from [PZ99] which showed that replace rules for definitions of predicates in conjunction with a propositional style prover outperformed Otter on some first-order problems involving definitions. These results provide further motivation for propositional techniques in first-order theorem proving.

Problem	OSHL Time (seconds)	Otter Time (seconds)	Clauses generated by Otter
P1	0.30	0.03	51
P2	2.30	1000.00+	41867
P3	11.25	1000.00+	41867
P4	1.35	1000.00+	27656
P5	2.00	1000.00+	54660

Table 1.3: Comparison of OSHL using replacement rules and Otter on 5 problems involving definitions quoted from [PZ99].

Ordered semantic hyperlinking (OSHL) [PZ00] is a theorem prover for first-order logic based on propositional techniques. OSHL solves first-order logic problems by combining an efficient propositional decision procedure and a strategy for instantiating first-order logic to propositional logic. This dissertation explores propositional approaches to theorem proving and how much first-order theorem proving one can do without true unification. It formulates some techniques that are shown to enhance the performance of the OSHL theorem prover. It shows that OSHL, thus enhanced and implemented as the prover OSHL-U, performs better than resolution on some categories of problems; these categories of problems are shown to belong to the classes of problems that are hard for resolution. The techniques applied to enhance OSHL performance are applicable to other theorem provers, too, and contribute towards building more powerful theorem provers, in general.

1.2 Thesis Statement

The use of syntactic strategies, heuristics and semantics significantly increases the power of OSHL, an instance-based automated theorem prover for first-order logic, and, on certain kinds of problems, produces performance comparable and even superior to that of resolution-based provers.

1.3 Summary of Results

Logical rules of inference and heuristics that control these rules are some techniques of guiding an automated theorem prover to make educated guesses at which part of the search space is more likely to contain a proof. So rules of inference and heuristics are an important factor in a theorem prover's success. This thesis extends the OSHL strategy with rules and heuristics and implements these as a theorem prover, OSHL-U. OSHL-U is so named because the inference rules it adds to OSHL are called *unit rules* or *U rules*. OSHL-U demonstrates that the addition of these rules leads to an improvement in performance of the OSHL strategy,

obtaining over 4 times as many proofs on problems from the Thousands of Problems for Theorem Proving (TPTP) repository version 2.5.0 as with only OSHL rules in the same amount of CPU time. The TPTP provides benchmark problems for evaluating performance of automated theorem proving systems. OSHL-U is also compared to resolution-based strategies of the theorem prover Otter [McC94]. On many problems, OSHL-U produces search spaces significantly smaller than those produced by Otter and uses less storage space than Otter. In terms of CPU time, OSHL-U obtains more than half the number of proofs on TPTP 2.5.0 problems as does Otter in the same amount of CPU time; in the categories of Field Theory (FLD) and Set Theory (SET) problems, OSHL-U obtains more proofs than Otter. These results are remarkable because the OSHL-U implementation is rather naive and lacks the sophistication of Otter such as term rewriting, equality handling, and data structures for efficiency. On the same processor, while Otter is capable of producing clauses in the order of 10,000 per second, OSHL-U only produces clauses in the order of 100 per second. Despite a slower inference rate, OSHL-U is already superior to Otter on FLD and SET problems.

Inference efficiencies of Otter and OSHL-U are also compared independent of their inference rate. These experiments show that OSHL-U is more time-efficient and more space-efficient on non-Horn problems, while Otter is more time-efficient and space-efficient on Horn problems.

Often, problems require expanding definitions in order to be proved. However, definition expansion increases the search space and can cause a theorem prover to get lost in unnecessary computation. How a theorem prover handles definitions is, therefore, important. OSHL-U and several leading theorem provers such as Vampire [RV99] and E-SETHEO [SW00] are tested on sets of problems that require expanding definitions of predicates in order to obtain proofs. Often, a predicate p is expressed in terms of other predicates in a clause, and such clauses are referred to as the definition of predicate p . In proving theorems, it is sometimes necessary to replace a predicate by its definition and this is referred to as definition expansion. Definition expansion leads to large clauses and large numbers of clauses. Indiscriminate expansion of definitions can quickly cause a theorem prover to generate many clauses unnecessary for the proof, and thus affect theorem prover efficiency. Hence it is important for provers to know when to expand definitions and when not to. OSHL-U, though a general theorem prover without special rules for expanding definitions, demonstrates superior performance to all the other provers.

Human mathematicians deal well with inference control owing to their knowledge of the problem domain, and their experience with solving similar problems. Automated theorem provers empowered with some degree of such expert knowledge have the potential to perform much better than without such knowledge. This is investigated in my thesis by using semantics to guide OSHL-U on some problems. The results indicate that use of semantic guidance can

indeed make the proof search more efficient; the use of semantics helps to obtain proofs of some problems that otherwise cannot be obtained with OSHL-U and on some problems produces smaller search spaces than without such semantic guidance.

The thesis tries to address the question of how essential true unification is to theorem proving for first-order logic, and how much theorem proving one can do without it. It tries to understand what characterizes problems that are hard for resolution, and shows that propositional methods are more powerful than resolution on such problems. This contributes to enhancing the power of automated theorem provers for first-order logic overall.

1.4 Outline of Thesis

This chapter provided a general overview of the thesis. Chapter 2 gives some background on automated theorem proving in first-order logic and provides definitions and terminology that is used throughout the dissertation. Chapter 3 first briefly describes the OSHL strategy; then it describes the inference rules and algorithms that extend OSHL and their implementation in the theorem prover, OSHL-U. In Chapter 4, we give experimental comparisons of execution time efficiency for OSHL-U and Otter. Chapter 5 describes refinements to the OSHL-U implementation that resulted in an increase in the number of proofs. Chapter 6 compares the space efficiency of OSHL-U and Otter search strategies using search spaces and storage spaces. Chapter 7 discusses techniques for comparing theorem provers independent of their inference rate and compares OSHL-U and Otter using these techniques. Chapter 8 compares OSHL-U to several leading automated theorem provers on sets of problems that require definition expansion for proofs. Chapter 9 deals with the use of semantic guidance for improving search efficiency of OSHL-U. Chapter 10 summarizes the conclusions of the thesis. Some of the results in Chapter 4 and Chapter 6 have appeared in [DP03] and [MP05].

Chapter 2

Background on First-Order Automated Theorem Proving

First, we present some notation and terminology for first-order theorem proving that will be used throughout this thesis. Then we define the theorem proving problem, and the input format that is used for such problems on the theorem provers described here. We provide some background on resolution, a few more definitions that use the concept of resolution, and some background on propositional methods of theorem proving in first-order logic. We also describe the TPTP repository, which provides the problems that are used for the experiments in this dissertation.

2.1 General Terminology and Definitions

In this section, we introduce some concepts related to our future discussions. A general background in first-order logic and refutational theorem proving is assumed. For a general introduction to these areas, we refer the reader to [Fit96]. A *term* is a well-formed expression composed of variables, constant symbols and function symbols, such as $f(X, g(a))$. In this thesis, variable symbols always begin with uppercase, such as X and Set , and constant symbols and function symbols begin with lowercase, such as a and $f(\dots)$. An *atom* is an expression of the form $p(t_1, \dots, t_n)$, where t_i for $i \in [1, n]$ are terms and p is a predicate symbol. A *literal* is an atom (*positive literal*) or an atom preceded by a negation sign (*negative literal*). A *clause* is a disjunction of a set of literals and may be written using the disjunction symbol \vee as $p(X) \vee \neg q(Y, Z)$ or in set notation as $\{p(X), \neg q(Y, Z)\}$. A *unit clause* is a clause containing only one literal. A *Horn clause* is a clause containing at most one positive literal. A *Horn set* is a set of Horn clauses.

A *substitution* is a mapping from variables to terms that is the identity on a finite number of variables. Informally speaking, a substitution tells us to replace variables by corresponding

terms for certain variables in a term, literal or clause; any other variable is considered to be replaced by itself. For example, if L is a literal and α a substitution, then $L\alpha$ is the literal resulting from replacing the variables in L by their image under α . Substitutions can be applied to terms and clauses in a similar way. M is called an *instance* of a literal L , if there is a substitution α such that $L\alpha = M$. A literal (or clause) is considered to be *fully instantiated* if it contains no variables. A fully instantiated literal (or clause) is also referred to as a *ground* literal (or clause). A clause, not necessarily ground, obtained by applying a substitution to a clause C is referred to as an *instance* of C .

An *interpretation* I consists of a domain D and a set of mappings as follows. Constants and functions are mapped to elements and functions, respectively, defined in the domain. Predicates are relation mappings from the elements in the domain to truth values (true or false). These mappings assign meaning to the constants, functions, and predicates. For example, a problem consisting of the constants *adam*, *bob*, *christine*, and *diane*, the function *likes*(X), and the predicates *male*(X), *female*(X) could have an interpretation I consisting of domain $D = \{a, b, c, d\}$, where a, b, c, d are the domain elements, and the following set of mappings. A constant is mapped to an element in D , a function is defined by mappings from domain elements (which are the arguments to the function) to domain elements and a predicate is defined by mappings from tuples of domain elements (which are the arguments to the predicate) to truth values.

adam	$\mapsto a$
bob	$\mapsto b$
christine	$\mapsto c$
diane	$\mapsto d$

likes :	$D \mapsto D$
	$a \mapsto c$
	$b \mapsto d$
	$c \mapsto b$
	$d \mapsto a$

male :	$D \mapsto \{\text{true}, \text{false}\}$
	$a \mapsto \text{true}$
	$b \mapsto \text{true}$
	$c \mapsto \text{false}$
	$d \mapsto \text{false}$

female :	$D \mapsto \{\text{true}, \text{false}\}$
----------	---

$a \mapsto \text{false}$
 $b \mapsto \text{false}$
 $c \mapsto \text{true}$
 $d \mapsto \text{true}$

An interpretation or *semantics* is a *model of* or *models* a clause (or literal) if the clause (or literal) is true under the interpretation. That interpretation I models clause C can be written as $I \models C$. The *empty clause*, $\{\}$, is false under every interpretation. $I_0[L_1L_2 \dots L_n]$ denotes an interpretation that is identical to the interpretation I_0 except that the literals $L_1, L_2, \dots L_n$ are interpreted as true.

2.2 Problem Description

A problem in theorem proving is described by a set of first-order logic formulae called statements. The statements can be systematically transformed into clause form as described in the next section. *Axioms* are statements that define a theory and a statement to be proved is called the *conjecture* (if validity of the statement is not known a priori) or *theorem* (if it is known to be a valid statement). Theorem proving involves showing that the conjecture or the theorem is a logical consequence of the set of axioms. A *formal proof* consists of a sequence of statements leading up to the statement to be established as true. Each statement in the sequence is either an axiom, or follows from previous statement(s) in the sequence by applying a logical *rule of inference*.

The method of proof by contradiction shows that the negation of the theorem and the set of axioms are unsatisfiable by deriving the empty clause (or False) from the set of statements; this indicates that the set of statements somehow contradict each other. Because the set of axioms is satisfiable, the unsatisfiability arises due to the negated theorem. Thus the contradiction proves the theorem. Such a proof obtained by contradiction is called a *refutation proof*. In the rest of this dissertation, a proof will refer to a refutation proof, unless otherwise stated. An automated theorem prover is a fully automatic computer program that generates a proof of the conjecture or theorem, given the axioms and the negation of the conjecture or theorem.

2.3 Conjunctive Normal Form for Representing First-Order Logic Formulae

In this thesis, a problem is presented to a theorem prover in a normalized format called the Conjunctive Normal Form(CNF). A knowledge base in CNF is assumed to be a conjunction

of statements, hence the name. Each statement is a clause, that is, a set of disjunctions of literals. For example, a problem P could be expressed in CNF as the conjunction of statements S_1 , S_2 , and S_3 , written as $S_1 \wedge S_2 \wedge S_3$, or simply as

S_1 .

S_2 .

S_3 .

The \wedge symbol is dropped and is implicitly assumed. Each statement, S_i , is a disjunction of literals, $L_1 \vee L_2 \vee \dots L_n$ for some n , and this can also be written in set notation as $\{L_1, L_2, \dots, L_n\}$.

The input format is a restricted syntax of first-order logic. Each clause is first normalized to the Skolem Normal Form, in which all quantifiers appear at the beginning of the formula and all existential quantifiers precede all universal quantifiers. All existential quantifiers are then removed by replacing existentially quantified variables by Skolem functions. Then all the universal quantifiers are dropped and the variables remaining in the clause are implicitly assumed to be universally quantified.

Any arbitrary first-order formula can be transformed systematically to CNF representation by the following steps. Each of the steps is explained in brief. For a more detailed explanation, see [Fit96].

- Move negations in to the atoms.
- Remove existentially quantified variables by replacing with Skolem functions.
- Move universal quantifiers to the front.
- Convert the matrix of the formula to a conjunction of disjunctions.
- Remove universal quantifiers and Boolean connectives.

Negations are moved in by the following set of rewrite rules.

$$\begin{aligned}
 \neg (\text{forall } x)A &\rightarrow (\text{exists } x)(\neg A) \\
 \neg (\text{exists } x)A &\rightarrow (\text{forall } x)(\neg A) \\
 (A \supset B) &\rightarrow ((\neg A) \vee B) \\
 \neg (A \wedge B) &\rightarrow ((\neg A) \vee (\neg B)) \\
 \neg (A \vee B) &\rightarrow ((\neg A) \wedge (\neg B)) \\
 \neg (\neg A) &\rightarrow A
 \end{aligned}$$

Before replacing existentially quantified variables by Skolem functions, common variables in a formula are renamed so that each variable appears in only one quantifier. Existential quantifiers are eliminated by replacing formulas of the form $(\text{exists } x)A[x]$ by $A[f(x_1, \dots, x_n)]$ where x_1, \dots, x_n are all the universally quantified variables whose scope includes the formula A , and f is a new function symbol that does not already appear in the formula. Removal of existential quantifiers in this way is called skolemization and the new function symbols, such as f , are called Skolem functions.

Universal quantifiers are moved to the front of the formula by the following rewrite rules.

$$\begin{aligned} ((\text{forall } x)A) \vee B &\rightarrow (\text{forall } x)(A \vee B) \\ B \vee ((\text{forall } x)A) &\rightarrow (\text{forall } x)(B \vee A) \\ ((\text{forall } x)A) \wedge B &\rightarrow (\text{forall } x)(A \wedge B) \\ B \wedge ((\text{forall } x)A) &\rightarrow (\text{forall } x)(B \wedge A) \end{aligned}$$

The matrix of the formula is converted to a conjunction of disjunctions by applying the following rewrite rules. Note that we are losing multilevel structure here.

$$\begin{aligned} (A \vee (B \wedge C)) &\rightarrow (A \vee B) \wedge (A \vee C) \\ ((B \wedge C) \vee A) &\rightarrow (B \vee A) \wedge (C \vee A) \end{aligned}$$

The only remaining quantifiers are universal quantifiers. These are dropped, and all variables are implicitly assumed to be universally quantified. Boolean connectives are replaced to convert a conjunctive normal form formula of the form

$$(A_1 \vee A_2 \vee \dots \vee A_i) \wedge (B_1 \vee B_2 \vee \dots \vee B_j) \wedge \dots \wedge (C_1 \vee C_2 \vee \dots \vee C_k)$$

to the form

$$\begin{aligned} \{A_1, A_2, \dots, A_i\}. \\ \{B_1, B_2, \dots, B_j\}. \\ \{C_1, C_2, \dots, C_k\}. \end{aligned}$$

As an example, consider the following first-order logic statement that expresses the subset axiom in set theory; ‘member’ and ‘subset’ are the predicates.

$$(\text{forall } \textit{Subset}) (\text{forall } \textit{Superset}) (\text{subset}(\textit{Subset}, \textit{Superset}) \vee (\text{exists } \textit{SomeElement}) (\text{member}(\textit{SomeElement}, \textit{Subset}) \wedge \text{not member}(\textit{SomeElement}, \textit{Superset})))$$

The statement is normalized and expressed as:

$$\text{subset}(\textit{Subset}, \textit{Superset}) \vee \text{member}(f(\textit{Subset}, \textit{Superset}), \textit{Subset}).$$

$subset(Subset, Superset) \vee not\ member(f(Subset, Superset), Superset).$

Here, f is a Skolem function that removed the existentially quantified variable *SomeElement*. The normalized form consists of the conjunction of two clauses.

To be input to the theorem prover, the formula is expressed in set notation as:

$\{subset(Subset, Superset), member(f(Subset, Superset), Subset)\}.$
 $\{subset(Subset, Superset) \vee not\ member(f(Subset, Superset), Superset)\}.$

A set of clauses expressed in CNF is unsatisfiable if for every interpretation I , there exists a clause that is False in I . A set of clauses in CNF is satisfiable if there exists an interpretation I in which all the clauses in the set are True.

2.4 Herbrand Sets and Herbrand's Theorem

A kind of interpretation that is particularly interesting for automated theorem proving is the *Herbrand interpretation*. A Herbrand interpretation is defined relative to a set S of clauses as follows. A *Herbrand Universe* for S is the set of all ground terms that can be formed from the constant and function symbols in S ; if S has no constant symbols, then an extra constant symbol is added. A Herbrand interpretation is an interpretation where the domain is a Herbrand universe and where each constant and function symbol is interpreted as itself.

If S is a set of clauses, then a *Herbrand set* for S is an unsatisfiable set T of ground clauses such that for every clause D in T , there is a clause C in S such that D is an instance of C . The well-known Skolem-Herbrand-Goedel theorem, also known as Herbrand's theorem [CL71], states that a set S of clauses is unsatisfiable iff there is a Herbrand set T for S .

2.5 Propositional Approach to First-Order Theorem Proving

Propositional methods applied to first-order theorem proving are based on enumerating a Herbrand set for the set of first-order clauses. The idea of exhaustive enumeration can be explained as follows.

Suppose S is a set of first-order clauses. Then let $Herb_S$ be the set of ground instances of clauses in S , such that all symbols appearing in clauses in $Herb_S$ appear in S , except that if S has no constant symbol, then an additional constant symbol is added in $Herb_S$. Let C_1, C_2, \dots be an enumeration of clauses in $Herb_S$, and let the set $\{C_1, C_2, \dots, C_i\}$ be denoted by T_i . Then the procedure for exhaustive enumeration is as shown in Algorithm 1.

Algorithm 1 Procedure for exhaustive enumeration of Herbrand set for set of clauses S

procedure enumerate(S)

```

1: let  $C_1, C_2, \dots$  be an enumeration of clauses in the Herbrand set for  $S$ 
2:  $i = 1$ 
3: repeat
4:   let the set  $\{C_1, C_2, \dots, C_i\}$  be denoted by  $T_i$ 
5:   test  $T_i$  for unsatisfiability
6:   if  $T_i$  is unsatisfiable then
7:     return "unsatisfiable"
8:   end if
9:    $i = i + 1$ 
10: until  $T_i$  is not equal to  $T_{i-1}$ 
11: return "satisfiable"

```

end enumerate

The procedure `enumerate(S)` enumerates progressively larger Herbrand sets for S , denoted by T_i , and tests them for unsatisfiability in 5. If S is unsatisfiable, the procedure will eventually return "unsatisfiable" 7. Otherwise, if $Herbs$ is finite then the condition in 10 will eventually be satisfied, and the procedure will return "satisfiable" 11. Otherwise the procedure will run forever. In this sense, the procedure is *complete* and this is a direct application of Herbrand's theorem.

2.5.1 Davis-Putnam-Logemann-Loveland (DPLL) Procedure

The first to apply the blind enumeration algorithm to clause form first-order theorem proving was [DLL62]; this paper used a slightly refined form of the original Davis-Putnam procedure [DP60], a reasonably efficient decision procedure, to test the sets T_i for unsatisfiability. Others had used enumeration procedures before this, but none had used Skolem functions together with clause form and none had used as efficient a propositional decision procedure. The original Davis-Putnam procedure is not used much nowadays; instead what is usually used is this later refinement due to David, Logemann and Loveland called DPLL. The DPLL test for unsatisfiability has three main components.

- Transformation to conjunctive normal form
- Rules for simplification
- Splitting

The conjunctive normal form has been discussed. Simplification consists of two main rules.

- The 1-literal rule. If a unit clause $\{L\}$ appears in the problem, remove all clauses that contain L , and remove all occurrences of $\neg L$ from the other clauses. This is equivalent

to assigning a truth valuation of True to L . It reduces the number of literals and the resulting problem is unsatisfiable iff the original problem is unsatisfiable.

- The affirmative-negative rule. If a literal L occurs only negated (that is, there are no clauses containing L) or only unnegated (that is, there are no clauses containing $\neg L$), then remove all clauses containing $\neg L$ or L . This is equivalent to assigning a truth valuation of False (True) to L . It reduces the number of literals and the resulting problem is unsatisfiable iff the original problem is unsatisfiable.

Splitting can be described as follows. Suppose the set of clauses S has some clauses containing a literal L and some clauses containing $\neg L$. Let S_L be the set of clauses resulting from S by removing all clauses containing L and deleting all occurrences of $\neg L$ from the other clauses. Similarly, let $S_{\neg L}$ be the set of clauses resulting from S by removing all clauses containing $\neg L$ and deleting all occurrences of L from the other clauses. Then replace S by the two new clause sets S_L and $S_{\neg L}$. This does a case analysis on L by splitting the tree of possible truth assignments for the variables into two subtrees where L is assigned True and False respectively. This is referred to as *splitting on the literal L* .

2.6 The Resolution Approach

Robinson [Rob65] developed the resolution procedure which avoids exhaustive enumeration of the Herbrand set by use of a technique called unification. Many other strategies developed since also use unification in various ways.

2.6.1 Rule of Resolution

Resolution, also referred to as *binary resolution*, takes two statements and resolves them into one. In a simplified form, the inference rule of resolution can be described as follows.

$$\frac{(A \vee B), (\neg B \vee C)}{(A \vee C)}$$

2.6.2 Unification

In applying the rule of resolution to first-order logic statements, variables often have to be matched to check that the inference is allowed. For example, consider the following two statements.

S_1 : $\neg \text{equal_sets}(\text{Subset}, \text{Superset}) \vee \text{subset}(\text{Subset}, \text{Superset})$
 S_2 : $\text{equal_sets}(b, bb)$

To apply the rule of resolution to S_1 and S_2 , the literal `equal_sets(b, bb)` in S_2 has to be matched up with the literal `equal_sets(Subset, Superset)` in S_1 ; for this the variable `Subset` is matched to the constant `b`, and the variable `Superset` is matched to the constant `bb`. Such a matching is referred to as unification.

As an example of when such matching is not allowed, consider the following two statements.

S_3 : $\neg \text{member}(\text{member_of_1_not_of_2}(\text{Subset}, \text{bb}), \text{bb}) \vee \text{subset}(\text{Subset}, \text{bb})$

S_4 : `subset(b, b)`

To apply the rule of resolution to S_3 and S_4 , the literal `subset(Subset, bb)` in S_3 has to be matched to the literal `subset(b, b)` in S_4 . This requires matching constant `bb` to constant `b`, which is not allowed. So the literals `subset(Subset, bb)` and `subset(b, b)` do not unify and the rule of resolution can not be applied to S_3 and S_4 .

Matching is not allowed when one of the terms being matched occurs in the other term. For example, the term x occurs in the term $f(x, y)$ and therefore these two terms cannot be unified. This is referred to as the *occurs-check problem* in unification.

A *unifier* of two terms (or literals, or clauses), L and M , is a substitution θ such that $L\theta$ and $M\theta$ are identical. A *most general unifier* of two terms (or literals, or clauses), L and M , is a substitution θ such that for any other unifier α of L and M , there is a substitution ϕ such that $L\theta\phi$ and $L\alpha$ are identical. ($L\theta\phi$ is obtained by applying substitution ϕ to the result of applying substitution θ to L .) For example, a most general unifier of $p(a, X, f(g(Y)))$ and $p(Z, f(Z), f(U))$ is $\{Z \leftarrow a, X \leftarrow f(a), U \leftarrow g(Y)\}$.

Algorithm 2, which is quoted from [Pla99], gives a simple algorithm to find the most general unifier of two terms r and s . In the procedure `Unify`, if the term r is a variable 1, then if the term s is syntactically identical to r 2, the terms are already the same and the procedure returns the empty substitution; otherwise, the occurs-check test is performed 4 to check whether r occurs in the term s . If r occurs in s , then the unification fails; otherwise, r is substituted by the term s to unify the terms 7.

If r is not a variable and s is a variable 9, then an occurs-check is performed to test whether s occurs in r . If s occurs in r , then the unification fails; otherwise, s is substituted by the term r to unify the terms 13.

If neither r nor s is a variable, then the top-level function symbol of the two terms are compared. If the function symbols are different 15, then the unification fails. If the function

Algorithm 2 Unify two terms r and s

procedure Unify(r, s)

```

1: if  $r$  is a variable then
2:   if  $r \equiv s$  then
3:     return  $\{\}$ 
4:   else if  $r$  occurs in  $s$  then
5:     return fail
6:   else
7:     return  $\{r \mapsto s\}$ 
8:   end if
9: else if  $s$  is a variable then
10:  if  $s$  occurs in  $r$  then
11:    return fail
12:  else
13:    return  $\{s \mapsto r\}$ 
14:  end if
15: else if the top-level function symbols of  $r$  and  $s$  differ or have different arities then
16:  return fail
17: else
18:  suppose  $r$  is  $f(r_1 \dots r_n)$  and  $s$  is  $f(s_1 \dots s_n)$ 
19:  return (Unify_lists( $[r_1 \dots r_n], [s_1 \dots s_n]$ ))
20: end if

```

end Unify

procedure Unify_lists($[r_1 \dots r_n], [s_1 \dots s_n]$)

```

1: if  $[r_1 \dots r_n]$  is empty then
2:   return  $\{\}$ 
3: else
4:    $\theta \leftarrow \text{Unify}(r_1, s_1)$ 
5:   if  $\theta \equiv \text{fail}$  then
6:     return fail
7:   end if
8:    $\alpha \leftarrow \text{Unify\_lists}([r_2 \dots r_n]\theta, [s_2 \dots s_n]\theta)$ 
9:   if  $\alpha \equiv \text{fail}$  then
10:    return fail
11:  end if
12: end if
13: return  $\{\theta \circ \alpha\}$ 

```

end Unify_lists

symbols are identical, then the list of arguments of the two terms are attempted to be unified by the procedure `Unify_lists` 19.

In the procedure `Unify_lists`, if the two lists are empty 1, no unification is necessary, so the empty substitution is returned. Otherwise, unification is attempted on the the first terms occuring in the lists 4. If this fails, then `Unify_lists` fails 6. Otherwise, the substitution unifying the first occuring terms is θ , and `Unify_lists` is recursively called on the remaining lists 8. If the recursive call fails, i.e., the remaining lists can not be unified, then `Unify_list` fails 10. Otherwise, the substitution unifying the remaining lists is α . In 13, \circ denotes *composition* of two substitutions, which is defined by $t(\theta \circ \alpha) = t\theta\alpha$, where t is a term. Composition of two substitutions is also a substitution. $\theta \circ \alpha$ is returned by `Unify_lists`.

To extend the unification of terms to unification of literals L and M , note that if L and M have different signs or different predicate symbols, then the unification fails. Otherwise, suppose L and M are $P(l_1, l_2, \dots, l_n)$ and $P(m_1, m_2, \dots, m_n)$, respectively (or their negations). Then the most general unifier of L and M is `Unify_lists` ($[l_1, l_2, \dots, l_n], [m_1, m_2, \dots, m_n]$).

The unification algorithm is similar to that of Robinson [Rob65]. This algorithm, though it takes exponential time on large terms, is often efficient in practice. Unification algorithms that are very efficient and take linear time on large terms have been devised.

2.6.3 Resolution Theorem Proving

Resolution theorem proving uses the resolution rule to obtain proofs by contradiction. The initial knowledge base consists of the axioms and the negation of the theorem to be proved. The proof technique involves maintaining a knowledge base of statements — all assumed to be true — and repeatedly resolving two statements and adding the resulting statement, the *resolvent*, to the set of statements comprising the knowledge base. When two statements are resolved to produce the empty clause or False, the theorem is established to be True. A proof can be produced by backtracking from the empty clause and building a chain of inferences backwards to the axioms in the initial knowledge base.

In practice, several other strategies and refinements of resolution may be used to make a theorem prover more efficient. Some of these are *hyper-resolution*, *demodulation* and *set of support*. Detailed descriptions of these and other strategies and refinements can be found in [BG01]. Hyper-resolution is a refinement of resolution that restricts the inferences that are performed and performs a sequence of such resolutions in one step, thus reducing the number of intermediate results to be stored. Demodulation replaces equals by equals, permitting

simplification of expressions. The set of support strategy forces all resolutions to involve a statement from a specified set (the set of support) or a statement derived from the set.

2.6.4 Further Definitions

A *unit resolution* is a resolution in which at least one parent clause is a unit clause. For example, resolution of $\{a, b, c\}$ with $\{\text{not } a\}$ to produce $\{b, c\}$ is a unit resolution.

A *UR resolution* (unit resultant resolution) is a sequence of resolutions with unit clauses and a non-unit clause to produce a unit clause. For example, a sequence of resolutions of the unit clauses $\{p\}$, $\{\text{not } q\}$, $\{r\}$ with the non-unit clause $\{\text{not } p, q, \text{not } r, s\}$ to produce the unit clause $\{s\}$ is a UR resolution. Otter [McC90], a well-known first-order resolution prover, also includes a sequence of resolutions with unit clauses and a non-unit clause to produce the empty clause (which has no literals) in its definition of UR resolution. This makes UR resolution a complete strategy in Otter.

2.7 Theorem Proving in First-Order Logic

Early theorem proving strategies (such as that of Gilmore [Gil60]) were based on the idea of instantiating a set of first-order clauses to obtain a set of propositional clauses, and then applying a propositional decision procedure to test satisfiability. Some recent provers such as SATCHMO [MB88] and Baumgartner's first-order DPLL method [Bau00] continue in this tradition. Ideas have also been proposed for enhancing instance-based theorem proving systems by employing decision procedures for first-order fragments more complex than propositional logic [GK03]. However, since Robinson's groundbreaking paper on resolution [Rob65] and Loveland's work on model elimination [Lov68, Lov69], the focus of the field has largely shifted to these and other similar approaches. Despite their successes, a shortcoming of such strategies, in a fully automated mode, is their weakness on non-Horn problems. This thesis uses OSHL to investigate the propositional approach to proving first-order theorems, especially those that are difficult for resolution. OSHL is based on exhaustive instantiation of the Herbrand universe and does not use true unification between non-ground literals. The objective is to understand how essential true unification is to theorem proving and how much theorem proving can be done without it.

In Section 2.5.1, we described the DPLL technique that is the basis of all propositional-style provers for first-order logic. Now we will briefly mention some of the renowned theorem provers for first-order logic based on propositional techniques.

2.7.1 The First-order DPLL Procedure

The DPLL procedure was originally devised as a proof-procedure for first-order logic. However, it has been used very successfully for propositional logic. Some of the most successful propositional satisfiability (SAT) solvers to date are based on DPLL. However, DPLL's treatment of quantifiers, based on instantiation into ground formulas, makes it far less efficient for first-order logic.

The more recent first-order DPLL (FDPLL) calculus by Baumgartner [Bau00] was the first successful attempt to lift the DPLL procedure to the first-order level without resorting to ground instantiation. FDPLL lifts to the first-order case the core of the DPLL procedure, the splitting rule, but ignores other aspects of the procedure. These other aspects are not necessary for completeness; however, they are crucial for the effectiveness of DPLL in practice. FDPLL uses a new technique to represent first-order interpretations, where a literal specifies truth values for all its ground instances unless there is a more specific literal specifying opposite truth values; in that case, the specific literal overrides the more general literal.

The splitting rule in DPLL essentially carries out a case analysis with respect to a propositional variable A , i.e., the current clause set S splits into two cases — one where A is “true” and one where A is “false” — and this leads to further simplification. FDPLL lifts this splitting to the first-order level by splitting on complementary non-ground literals like $P(x, y)$ and $\neg P(x, y)$. But the implicit universal quantification of the variables x and y as is the case in clause-form theorem proving leads to an unsound calculus. So the way a literal is read is modified in FDPLL as follows. A literal represents all its ground instances, for example, $P(x, y)$ represents instances $P(a, a)$, $P(a, b)$, $P(b, a)$ and $P(b, b)$ assuming that a and b are the only constants. However, a more specific instance of the literal with complementary sign, such as $\neg P(x, b)$, overrides the default interpretation of the more general literal. In this example, $P(x, y)$ and $\neg P(x, b)$ together represent $P(a, a)$, $\neg P(a, b)$, $P(b, a)$ and $\neg P(b, b)$. This can be associated to an interpretation. In this way, a “case” in FDPLL is a set of (possibly non-ground) literals that an interpretation can be associated to.

The splitting rule in FDPLL is explained as follows. Suppose $C\theta$ is an instance of C that is “false” in the interpretation I associated to the current case. Then a split is attempted with with a literal $L \in C\theta$ in order to *repair* I to an interpretation that assigns “true” to L , and, therefore, to $C\theta$. If this is not possible because of contradiction between $C\theta$ and the current case, then the current case is refuted; otherwise, splitting is performed on L and two new cases are created — one extending the current case with L and the other extending the current case with $\neg L$. FDPLL repeatedly carries out splits in this way until either every case is refuted signifying unsatisfiability, or no clause instance $C\theta$ is falsified by the current

interpretation signifying that the current case is a satisfiable model.

For a more detailed treatment of FDPLL, the reader is referred to [Bau00].

2.7.2 The Model Evolution Calculus

The Model Evolution (ME) calculus [BT03] provides a complete lifting of the DPLL procedure to first-order logic. The ME calculus borrows many fundamental ideas from and generalizes FDPLL, but it is an extension of DPLL rather than of FDPLL. The DPLL procedure provides a Herbrand model of the input formula whenever that formula is satisfiable; it generates this model incrementally as it progresses. The ME calculus lifts this model generation process to the first-order level. The goal of the ME calculus is to construct a Herbrand model of a given set Φ of clauses, if any such model exists. It does so by maintaining a finite set of literals called a *context*. The context C is a finite and compact representation of a Herbrand interpretation I_C serving as a candidate model for Φ . The interpretation I_C may not be a model of Φ because it does not satisfy some clauses in Φ . The main rules of the ME calculus serve to detect this situation and either repair I_C by modifying C so that it becomes a model of Φ , or recognize that I_C is unrepairable and fail. There are additional simplification rules in the calculus which, like in DPLL, simplify the clause set, hence speeding up the computation.

The ME calculus starts with a default candidate model that does not satisfy any positive literals and repairs it as needed until it becomes an actual model of the input clause set Φ , or until it is clear that Φ has no models at all. So all terminating derivations of a satisfiable clause set Φ produce a context C such that the corresponding interpretation I_C is a model of Φ . This provides counter-examples to invalid statements rather than only proving their invalidity.

In addition to being a more faithful lifting of the DPLL procedure, the ME calculus contains a more systematic treatment of *universal literals*, one of FDPLL's optimizations, and so has the potential of leading to much faster implementations.

For a detailed treatment of the ME calculus, the reader is referred to [BT03].

2.7.3 The Disconnection Theorem Prover

The Disconnection Theorem Prover (DCTP) [LS01] is an implementation of the disconnection tableau calculus. The system can also be used for model generation.

The disconnection tableau calculus integrates Plaisted's clause linking method [PL92] into a tableau control structure. The original clause linking method works by iteratively producing instances of the input clauses, which are occasionally tested for unsatisfiability by a separate propositional decision procedure. The use of a tableau as a control structure restricts the number of clause linking steps that may be performed. The tableau also provides a propositional decision procedure for the clause instances generated so there is no need for a separate propositional decision procedure. The disconnection tableau calculus consists of a single inference rule called the linking rule. In addition, a number of refinements are integrated into the system in order to improve the theorem prover's performance; these are — pruning of clause variants, pruning of redundant branches, unit simplification, unit lemma generation and unit subsumption. For a detailed treatment of the rules and refinements in DCTP, the reader is referred to [LS01].

2.7.4 Propositional Techniques at UNC

The propositional approach to first-order theorem proving has been explored at UNC-Chapel Hill by [AP92, PL92, Pla94, ZP97] leading up to the development of the Ordered Semantic Hyperlinking (OSHL) prover [PZ00]. OSHL is an attempt to apply propositional techniques to first-order logic. It is similar to DPLL, which has been shown to be more efficient than resolution on non-Horn propositional problems [PL92], suggesting that similar approaches might also be efficient for non-Horn first-order problems. However, one problem with the blind enumeration approach is that it enumerates many propositional instances that may not be needed for the proof.

OSHL differs from other provers that apply propositional techniques to first-order theorem proving such as clause linking [PL92], FDPLL [Bau00] and DCTP [LS01] in that OSHL works completely at the propositional level and that it does not perform true unification between non-ground literals. FDPLL does not work completely at the propositional level. DCTP also does not work completely at the propositional level, but employs true unification to generate instances. SATCHMO [MB88], though it has similarities to OSHL, does not use orderings or models in the way OSHL is capable of and also does not seem to have a counterpart for all of the U rules [DP03] added to OSHL in OSHL-U. The capability of OSHL for using sophisticated semantic guidance, that all these other provers lack, makes OSHL a unique system for study.

2.8 The TPTP Problem Set

TPTP stands for Thousands of Problems for Theorem Provers and is a repository of test problems that serves as a benchmark for automated theorem proving systems [SS98a]. The

version of TPTP used for the experiments in this dissertation is v2.5.0. TPTP v2.5.0 has first-order problems from 30 different domains. This thesis uses all the problems that are not known to be satisfiable from these 30 categories for tests; there are 4417 such problems.

Each problem in the TPTP comes with useful statistics such as number of non-Horn clauses in the problem, hardness rating and solvability status. When the number of non-Horn clauses is zero, the problem is a Horn problem. Otherwise, it is probably non-Horn; however, even if there are non-Horn clauses in the set, it is possible that a proof can be obtained using only Horn clauses.

TPTP ratings range from 0 to 1, with 0 being the easiest. The rating is based on how difficult the problem is for theorem provers that have been tested on the TPTP. A problem with a TPTP rating of 0 has been proved by all major automated theorem proving systems. A problem with a TPTP rating of 1 has not been proved by any automated theorem proving system.

The status of a problem could be *satisfiable* (known to be a satisfiable set of clauses), *unsatisfiable* (known to be an unsatisfiable set of clauses), *theorem* (a theorem, hence an unsatisfiable set of clauses), *open* (an open problem), and *unknown* (it is not known whether the set is satisfiable or unsatisfiable). The OSHL and OSHL-U strategy only detects unsatisfiability and does not guarantee termination on satisfiable sets. Therefore, this thesis does not use the problems with a satisfiable status for tests.

2.9 Strategy Selection and Performance Tuning in Modern Provers

Most modern theorem provers have been optimized for performance and use strategy selection and strategy scheduling. In addition, many provers have been tuned to perform well on the TPTP problem categories. We will look briefly at strategy selection and scheduling and performance tuning in different theorem provers. Then we justify our choice of a resolution theorem prover for experimental comparison to OSHL.

Strategy selection and strategy scheduling are implemented in many leading theorem provers today. Strategy selection refers to the analysis of a problem and the selection of a strategy to be used based on this analysis. Also, modern provers sometimes classify the problems into 60 or more categories and apply a different strategy to each category [Sch02]. Often, several potentially good strategies are identified and these are then scheduled to be applied in a certain order till a proof can be found [SW99, SW00, SS99, HJL99, RV02, Sch02];

this is referred to as strategy scheduling. Strategy selection and strategy scheduling make it difficult to run a single uniform strategy on all problems. It also becomes difficult to determine whether the performance of the prover is due to a single uniform strategy or the way in which the categories or strategies are selected.

In addition, the performance of modern provers is sometimes tuned to categories of problems in the TPTP. Based on pre-computed test results on the TPTP problems, the prover “learns” what strategy is effective for particular kinds of problems. The idea is similar to how human experts learn to identify problem-solving techniques based on prior experience and boosts prover performance. However, such performance tuning makes it difficult to separate out the performance of a prover due to the strategy used and the performance boost resulting from tuning on particular problem sets.

2.9.1 Choice of a Resolution Prover for Experimental Comparison

This dissertation compares OSHL-U with Otter, a first-order theorem prover based on the resolution-unification paradigm. Theorem provers such as SPASS [Wei97], GANDALF [Tam97], Vampire [RV99], and SETHEO [LSBB92] may perform better than Otter. However, these other provers have been thoroughly optimized and have efficient data structures, which give them a considerable advantage over OSHL and OSHL-U, which do not have such extensive optimizations. Some of these provers use case analysis (splitting), which is really a form of propositional reasoning imported into first-order logic. Otter is a useful prover for comparing the OSHL-U strategy to resolution because in the autonomous mode, Otter has a simpler and more uniform strategy selection [McC03]. Otter’s simple control structure allows to apply a single uniform strategy to all problems. Otter does not break the TPTP problems into categories in the way that many other provers do. Also, Otter has not been especially tuned to a particular problem set in the TPTP; when it uses strategy selection, it does so based on analysing only the problem syntax and not based on any pre-computed results. However, Otter’s efficient data structures still give it an execution speed advantage over OSHL-U, which is not using such data structures.

Chapter 3

Ordered Semantic Hyper-Linking with Unit Rules

A brief description of the OSHL algorithm is given, followed by a description of the OSHL-U rules of inference, algorithm and heuristics. OSHL-U is the contribution of this thesis; it builds on and extends OSHL.

In order to describe OSHL and OSHL-U, we need the notion of eligible literals. An *eligible literal* is a ground literal that is true under the current interpretation. Suppose I is an interpretation and C a ground clause such that $I \not\models C$. The interpretation could be modified to make it a model of C by making a literal L of C true under I . The modified interpretation is written as $I[L]$ and L is called an eligible literal under $I[L]$. As defined in Chapter 2, $I[L_1 \dots L_n]$ denotes an interpretation that is the same as I except that the literals L_i for $i \in [1, n]$, are satisfied by $I[L_1 \dots L_n]$; we will refer to L_1, \dots, L_n as the eligible literals.

3.1 The OSHL Strategy

The OSHL strategy begins with an initial interpretation and progresses in the search for a proof by generating ground instances of input clauses. Such ground instances are used to refine the interpretation. OSHL maintains a set of ground instances of input clauses such that the current interpretation is always a minimal model of this set. The two key steps in the OSHL algorithm are to generate a ground instance of an input clause D that interprets to false under the current interpretation, and then to modify the current interpretation so as to make D true. The ground instance generated is minimal in some ordering defined on clause instances. As was shown in [PZ00], picking the ground instance subject to the restriction of being minimal makes OSHL a complete theorem prover.

OSHL tries to determine if a set S of first-order clauses is unsatisfiable by constructing a set T of ground instances of clauses in S . It is assumed that if the set T is unsatisfiable,

then it contains the empty clause $\{\}$. It is assumed also that some ordering on interpretations and some ordering on ground clauses is given. The OSHL algorithm repeats the following sequence of operations till T contains $\{\}$.

- Construct interpretation I , a model of T , such that I is minimal in the ordering on interpretations
- Select a ground instance D of a clause in S such that $I \not\models D$ and D is minimal in the ordering on clauses subject to this restriction. Replace T by $T \cup D$
- Modify the set T to preserve or increase its minimal model

The OSHL strategy is based on an exhaustive enumeration of the Herbrand base. One problem with any enumeration strategy is that of efficiency. In OSHL, this problem is alleviated by the use of semantics in order to generate instances relevant to the proof with efficiency. The ordering on clauses that is commonly used is size-lexicographic, that is, smaller instances are generated before larger, and when two instances have the same size, the instance occurring first in lexicographic order (such as English alphabetic order) is generated first. The lexicographic order could also be specified by the user to be some other total order. The search, therefore, progresses in order of increasing size of terms; proofs with smaller terms are generated before those with larger terms. As was mentioned in [PZ00], OSHL had problems generating proofs with larger sized terms. OSHL also did not have any syntactic guidance in instance generation, and therefore, depended solely on the specified semantics for guidance.

3.2 OSHL extended with Unit Rules of Inference

The original intent in designing OSHL was to use it in conjunction with semantics that guide the proof search. So a deficiency of OSHL, in the absence of good semantics, is the blind enumeration of instances; this is constrained somewhat by the interleaving of instantiation and model searching and by the use of semantics, but it remains a problem. The rules of inference added to OSHL in this thesis are intended to provide better syntactic guidance in the enumeration of instances. The implementation of the resulting theorem prover is referred to as OSHL-U, which is OSHL extended with *unit rules* or *U rules*.

OSHL-U attempts to overcome the blind enumeration of instances in OSHL by making the generation of instances more intelligent when possible, while retaining the propositional character of OSHL. This is done by relaxing one of the constraints of OSHL, namely, that the instance generated must be a minimal instance contradicting a specified interpretation. OSHL-U permits the instance to be non-minimal and to contradict another interpretation, in exchange for avoiding the blind enumeration, when possible. OSHL-U uses a combination of

strategies including case analysis and unit resultant resolution on ground clauses. Only when none of these strategies are able to find an instance does OSHL-U resort to the enumeration strategy of OSHL. The proof of completeness of OSHL is based on the condition that the minimal model of the set of clauses T_{i+1} is larger than the minimal model of T_i for all i . The OSHL-U rules preserve the condition, so this also guarantees the completeness of OSHL-U.

OSHL-U consists of *basic rules* (from OSHL) and *U rules*. The basic rules by themselves are complete and work essentially at the ground level. The *U rules* are not necessary but perform operations involving unit clauses and frequently help to get proofs faster. Basic rules produce basic clauses while *U rules* produce *U clauses*. The OSHL-U rules operate on an *ascending sequence* which is a sequence $C_1 C_2 \dots C_n$ of ground clauses. Initially the ascending sequence is empty. The proof search stops when an ascending sequence containing the empty clause is derived; this indicates that S is unsatisfiable.

The ascending sequence consists of a (possibly empty) sequence of *basic clauses* followed by a (possibly empty) sequence of *U clauses*. The resolvent of two clauses, at least one of which is basic, is a basic clause and the resolvent of two *U clauses* is a *U clause*. Note that a *U clause* is not necessarily a unit clause (i.e., a clause with only one literal). Each clause in the ascending sequence has a selected literal which is the eligible literal; L_i denotes the eligible literal of clause C_i in the sequence. E is the set $\{L_1, L_2, \dots, L_n\}$ of eligible literals and \bar{E} is the set of their complements. Several rules add a clause C to the end of an ascending sequence; this is always subject to the restriction $C \cap E = \phi$ where E consists of the eligible literals before C is added.

There is a total syntactic ordering $<_{lit}$ on ground atoms that is extended to literals by $L <_{lit} M$ iff $at(L) <_{lit} at(M)$ where $at(\neg L) = L$ and if L is an atom, $at(L) = L$. This ordering restricts which literals may be selected from clauses. For basic clauses C_i , L_i is the $<_{lit}$ maximum literal in C_i . For *U clauses* C_i , if C_i contains a literal L that is not complementary to existing eligible literals, then some such L must be selected. Otherwise, a literal must be selected that is complementary to L_j for the maximum possible j .

The $<_{lit}$ ordering on literals is extended to a $<_c$ ordering on clauses as follows. Suppose L_1 and L_2 are the $<_{lit}$ -maximal literals in clauses C_1 and C_2 , respectively. Then $C_1 <_c C_2$ iff $L_1 <_{lit} L_2$. If L is the $<_{lit}$ -maximal literal in both C_1 and C_2 , then $C_1 <_c C_2$ iff $(C_1 - \{L\}) <_{lit} (C_2 - \{L\})$.

There is an ordering $<_{el}$ on the eligible literals. Suppose L_i and L_j are the selected literals from clauses C_i and C_j in the ascending sequence. Let i' be minimal such that L_i or $\neg L_i$ is the selected literal from clause $C_{i'}$, and similarly for j' . Then $L_i <_{el} L_j$ iff $i' < j'$.

I_0 is the initial interpretation. I_i models the set of clauses C_1, C_2, \dots, C_i and is obtained by refining I_0 to satisfy the eligible literals L_1, L_2, \dots, L_i of clauses C_1, C_2, \dots, C_i , respectively. So I_i can also be written as $I_0[L_1 \dots L_i]$.

3.2.1 Basic Rules in OSHL

Extension

Let $C\theta$ be a ground instance of an input clause C such that $C\theta \cap E = \phi$. Add $C\theta$ to the end of the ascending sequence and select a literal from it.

$$\frac{(C_1, C_2, \dots, C_n), C \text{ an input clause, } \theta \text{ a substitution, s.t.} \\ C\theta \text{ is ground and } <_c \text{ minimal in } S \text{ s.t. } I_n \not\models C\theta}{(C_1, C_2, \dots, C_n, C\theta)}$$

The rule of extension extends the ascending sequence by adding an instance. The added instance is generated from a clause C from the set of input clauses S by applying variable substitution θ to C . The instance, $C\theta$, interprets to false under the current interpretation I_n . Also, the instance $C\theta$ is selected such that, among all such instances possible, it is minimal in the $<_c$ ordering.

Resolution

If the selected literals L and M of the last two clauses C and D of the ascending sequence are complementary, remove these clauses and add their resolvent $(C - \{L\}) \cup (D - \{M\})$ to the ascending sequence.

$$\frac{(C_1, C_2, \dots, C_n, C_{n+1}), L \text{ is the selected literal in } C_n, \\ \neg L \text{ is the selected literal in } C_{n+1}}{(C_1, C_2, \dots, C_{n-1}, (C_n - \{L\}) \cup (C_{n+1} - \{\neg L\}))}$$

Clause Deletion

If $L_n < L_{n-1}$, delete C_{n-1} from the ascending sequence. The effect of clause deletion is to delete clauses whose selected literals are “out of order,” i.e., they violate the ascending order of the sequence. Resolution removes successive clauses whose selected literals have the same atom. If clause deletion and resolution are done as soon as possible, then after they are both finished, it will be true that for all i and j , $L_i <_{el} L_j$ iff $i < j$.

$$\frac{(C_1, C_2, \dots, C_{n-1}, C_n), (L_n < L_{n-1})}{(C_1, C_2, \dots, C_{n-1})}$$

3.2.2 U Rules

U Clause Deletion

This is like the basic rule of clause deletion but it applies to U clauses. It is also possible to apply U clause deletion if C_n is a U clause and C_{n-1} is a basic clause.

$$\frac{(C_1, C_2, \dots, C_{n-1}, C_n), (L_n < L_{n-1}), C_n \text{ is a } U \text{ clause}}{(C_1, C_2, \dots, C_{n-1})}$$

U Resolution

This is like resolution but applies to two U clauses.

$$\frac{\begin{array}{l} (C_1, C_2, \dots, C_n, C_{n+1}), C_n \text{ and } C_{n+1} \text{ are } U \text{ clauses,} \\ L \text{ is the selected literal in } C_n, \neg L \text{ is the selected literal in } C_{n+1} \end{array}}{(C_1, C_2, \dots, C_{n-1}, (C_n - \{L\}) \cup (C_{n+1} - \{\neg L\}))}$$

UR Resolution

Suppose C is a non-unit clause. Then a sequence of resolutions between C and unit clauses resulting in a unit clause is generally referred to as unit resultant resolution or UR resolution. The UR resolution rule in OSHL-U is a special case of UR resolution. It is described as follows.

Find $C \in S$ that gives L as a result of UR resolution with the set of eligible literals, E , such that L is a ground literal and $L \notin E$. Let $C\theta$ be an instance of C such that $C\theta \subset \overline{E} \cup \{L\}$. Add $C\theta$ to the end of the ascending sequence and select the literal L from it.

$$\frac{\begin{array}{l} (C_1, \dots, C_n), C = \{M_1, \dots, M_k, M\} \text{ an input clause, } \theta \text{ a substitution,} \\ M\theta \notin \{L_1, L_2, \dots, L_n\} \text{ and } C\theta \subset \overline{E} \cup \{M\theta\} \end{array}}{(C_1, C_2, \dots, C_n, C\theta[M\theta])}$$

This rule describes the derivation of a U clause instance that, immediately after being added to the ascending sequence, will cause a series of basic and/or U resolution and deletion. Every literal in $C\theta - \{M\theta\}$ is the complement of some eligible literal and, therefore, can be removed by an application of the Resolution or U Resolution rules with an existing clause in the sequence. Resolution and U Resolution rules (as well as Deletion and U Deletion rules) are applied to the sequence as soon as possible, so adding the clause $C\theta$ to the sequence ultimately generates the unit U clause, $\{M\theta\}$.

For example, given the sequence $(\{s(a), \mathbf{p(b)}\}, \{t(a), \mathbf{q(b)}\})$ and the clause $\{\text{not } p(X), \text{not } q(X), r(X)\}$, where the highlighted literals are the eligible literals, the sequence $(\{s(a), \mathbf{p(b)}\}, \{t(a), \mathbf{q(b)}\}, \{\text{not } p(b), \text{not } q(b), \mathbf{r(b)}\})$ is created.

Unit Filtering

Let D be obtainable from $C \in S$ by zero or more unit resolutions with unit clauses in S . Let $D\theta$ be an instance of D such that $D\theta \subset \overline{E}$. Add $D\theta$ to the end of the ascending sequence. Note that after this rule is done, either a resolution or a clause deletion (or their unit rule counterparts) will be possible, so the ascending sequence will get shorter. If $D = C$, this is called filtering.

$$\frac{\begin{array}{l} (C_1, \dots, C_n), \\ C = \{M_1, \dots, M_k, D\} \text{ an input clause, } \theta \text{ a substitution} \\ (\forall M_i \in C)(\neg M_i \in \{C_1, \dots, C_n\}) \text{ and } D\theta \subset \overline{E} \end{array}}{(C_1, \dots, C_n, D\theta)}$$

This rule describes how a U clause is added to the ascending sequence. As with the UR Resolution rule, an application of Unit Filtering will be followed by a series of basic and/or U resolutions and deletions. Every literal in $D\theta$ is the complement of some eligible literal. Therefore, unless at some point, the remaining clause (some subset of $D\theta$) is removed by U clause deletion, every literal will eventually be removed by the Resolution or U Resolution rules. When that happens, the empty clause will get added to the ascending sequence.

For example, given the sequence $(\{s(a), \mathbf{p(b)}\}, \{t(a), \mathbf{q(b)}\})$ and the clause $\{\text{not } p(X), \text{not } q(X)\}$, where the highlighted literals are the eligible literals, the sequence $(\{s(a), \mathbf{p(b)}\}, \{t(a), \mathbf{q(b)}\}, \{\text{not } p(b), \mathbf{\text{not } q(b)}\})$ is created.

Case Analysis

Let C be an input clause and L be a literal of C containing all the variables of C . Let θ be a substitution that causes L to become ground such that $L\theta \in \overline{E}$ and $C\theta \cap E = \phi$. Add $C\theta$ to the end of the ascending sequence.

$$\frac{\begin{array}{l} (C_1, \dots, C_n), C = \{M_1, \dots, M_k\} \text{ an input clause,} \\ j \in [1, k] \text{ s.t. every variable in } C \text{ occurs in } M_j \text{ and } \theta \text{ a substitution s.t.} \\ (M_j\theta \in \overline{E} \text{ and } C\theta \cap E = \phi) \end{array}}{(C_1, C_2, \dots, C_n, C\theta)}$$

The case analysis rule extends the ascending sequence with a U clause. Note that there are several pre-conditions for the case analysis rule to be applicable. First, all the variables in an input clause, C , occur in one of its literals, M_j , so that a substitution θ that instantiates all the variables in M_j also results in a fully instantiated clause instance $C\theta$. Second, $M_j\theta$ is a complement of an eligible literal. Also, no other literal in $C\theta$ is a complement of an eligible literal. When these conditions are satisfied, then a literal, $M_k\theta$ say, in $C\theta - \{M_j\theta\}$ could become a selected literal, or eligible to be true; this literal is a case. If subsequent UR resolutions and case analyses cause a refutation of this case (i.e., the literal, $M_k\theta$, is resolved away giving a new instance), then another literal from $C\theta - \{M_j\theta, M_k\theta\}$ becomes the selected literal. When every case has been refuted, the empty clause gets added to the ascending sequence.

For example, given the sequence $(\{s(a), p(b)\}, \{t(a), q(b)\})$ and the clause $\{\text{not } q(X), r(X), s(X)\}$, where the highlighted literals are the eligible literals, the sequence $(\{s(a), p(b)\}, \{t(a), q(b)\}, \{\text{not } q(b), r(b), s(b)\})$ is created by the case analysis rule.

3.2.3 Order of Applying Rules in OSHL-U

The order of applying the rules in the OSHL-U implementation is described by the flow of control that follows. The top level procedure is *oshlu_top(S)*, shown in Algorithm 3, where S is the set of input clauses, *empty_list* is the ascending sequence which is empty to start with, and *starting_size* gives the initial upper limit on the allowed size of generated instances. If S is a satisfiable set, then it is possible that *oshlu_top(S)* will run forever because OSHL-U does not necessarily detect satisfiability; otherwise *oshlu_top(S)* will eventually return a list containing the empty clause, signifying that a proof has been found.

As shown in Algorithms 3 and 4, the procedure *oshlu* invokes a procedure corresponding to each OSHL-U rule (basic or U) on an ascending sequence of clauses. In algorithm 3, each of these procedures returns “fail” if the corresponding rule does not apply, and otherwise returns the new ascending sequence that results from applying the rule. If the ascending sequence contains the empty clause (line 1), then a proof has been found and the procedure terminates by returning the ascending sequence (line 2). Otherwise, the Deletion rule is attempted to be applied first (line 4). If it succeeds (line 5), then the *oshlu* procedure is called recursively on the resulting ascending sequence (line 6); if it fails, then the Resolution rule is attempted (line 8). If Resolution succeeds (line 9), then the *oshlu* procedure is called recursively on the resulting ascending sequence (line 10). If neither Deletion nor Resolution succeeds, then the U rules are attempted by calling the procedure *try_U_rules* (line 12). If the U rules succeed (line 13), then the *oshlu* procedure is called recursively on the resulting ascending sequence (line 14); otherwise any clauses added by the U rules are thrown away

Algorithm 3 Order of applying rules in OSHL-U

procedure oshlu_top(S)

 return (oshlu(empty_list, starting_size, S))

end oshlu_top

procedure oshlu(clause_list , s , S) {call oshlu with size bound of s , ascending sequence S }

 1: **if** clause_list contains the empty clause **then**

 2: **return** clause_list

 3: **end if**

 4: $\text{new_list} \leftarrow \text{deletion}(\text{clause_list})$ {apply deletion rule}

 5: **if** $\text{new_list} \neq \text{fail}$ **then**

 6: **return** oshlu(new_list , s , S)

 7: **end if**

 8: $\text{new_list} \leftarrow \text{resolution}(\text{clause_list})$ {apply resolution rule}

 9: **if** $\text{new_list} \neq \text{fail}$ **then**

 10: **return** oshlu(new_list , s , S)

 11: **end if**

 12: $\text{new_list} \leftarrow \text{try_U_rules}(\text{clause_list}, s, S)$ {try U rules}

 13: **if** $\text{new_list} \neq \text{fail}$ **then**

 14: **return** oshlu(new_list , s , S)

 15: **end if**

16: {throw away U clauses if U rules fail}

 17: $\text{new_list} \leftarrow \text{extension}(\text{clause_list}, s, S)$ {apply extension rule}

 18: **if** $\text{new_list} \neq \text{fail}$ **then**

 19: **return** oshlu(new_list , s , S)

 20: **end if**

 21: **return** (oshlu(clause_list , $s+1$, S)) {increment size bound}

end oshlu

(line 16) and Extension is attempted (line 17). If Extension succeeds (line 18), then the *oshlu* procedure is called recursively on the resulting ascending sequence (line 19); otherwise the size bound is incremented by 1 and *oshlu* is called recursively with the incremented size bound (line 21). The initial value of size bound is passed into *oshlu* from *oshlu_top*. Subsequently, *oshlu* is called repeatedly with increasing values of size bound till it succeeds in finding a proof. If one were to regard the space of possible proofs as a tree where proofs using larger instances are further away from the root, then the progressively increasing values of the size bound ensure that the search progresses with iterative deepening. In other words, all proofs containing instances with a maximum size of n are generated before any proofs containing instances of size $n + 1$. Later on in the chapter, we will see several different ways in which size of an instance can be defined, and how OSHL-U computes the size of an instance.

Among the U rules, U deletion and U Resolution are performed as soon as possible and these shrink the ascending sequence. When it is not possible to apply these two rules anymore, the U rules that add clauses to the ascending sequence are attempted. The order in which they are attempted is Unit Filter first, followed by UR Resolution, and then Case Analysis.

Algorithm 4 shows the order in which the U rules are applied in the procedure *try_U_rules*. First, U Deletion is attempted (line 1). If it succeeds (line 2), then *try_U_rules* is called recursively on the resulting ascending sequence (line 3); otherwise U Resolution is attempted (line 5). If U Resolution succeeds (line 6), then if this results in removing all U clauses from the ascending sequence, i.e., all the clauses in the ascending sequence are basic clauses (line 7), then no more U rules are applied to the sequence. The procedure terminates and returns the ascending sequence. otherwise if there are U clauses still remaining in the ascending sequence, then *try_U_rules* is called recursively (line 10). If U Resolution fails, then Unit Filtering is attempted (line 14). If Unit Filter succeeds (line 15), then *try_U_rules* is called recursively on the resulting sequence (line 17); otherwise UR Resolution is attempted (line 19). If UR Resolution succeeds (line 20), then *try_U_rules* is called recursively on the resulting ascending sequence (line 22); otherwise Case Analysis is attempted (line 24). If Case Analysis succeeds (line 25), then *try_U_rules* is called recursively on the resulting ascending sequence (line 27); otherwise none of the U rules succeeded in generating an instance, so the procedure returns “fail” (line 29).

3.3 Examples of OSHL-U Operation

Short examples are given to illustrate how the OSHL-U prover strategy works. The first example problem is SET001-1 from the TPTP v2.5.0 [SS98a]. The problem is given in the CNF form described in Chapter 2. The theorem being proved states that a member of a set is also a member of that set’s supersets. The 9 input clauses, shown below, include 6 axiom

Algorithm 4 Order of applying U rules in OSHL-U

```

procedure try_U_rules(clause_list, s, S) {try U rules}
  1: new_list  $\leftarrow$  U_deletion(clause_list) {apply U deletion rule}
  2: if new_list  $\neq$  fail then
  3:   return try_U_rules(new_list, s, S)
  4: end if
  5: new_list  $\leftarrow$  U_resolution(clause_list) {apply U resolution rule}
  6: if new_list  $\neq$  fail then
  7:   if all clauses in new_list are basic clauses then
  8:     return new_list
  9:   else
  10:    return try_U_rules(new_list, s, S)
  11:  end if
  12: end if
  13: {recall that the resolvent of a basic clause and a unit clause is a basic clause}
  14: new_clause  $\leftarrow$  filter(clause_list, s, S) {see if unit filtering can find a contradiction clause}
  15: if new_clause  $\neq$  fail then
  16:   let new_list be clause_list with new_clause added to the end
  17:   return try_U_rules(new_list, s, S)
  18: end if
  19: new_clauses  $\leftarrow$  UR_resolution(clause_list, s, S) {do a round of ground UR resolution on
    all input clauses}
  20: if new_clauses  $\neq$  fail then
  21:   let new_list be clause_list with new_clauses added to the end
  22:   return try_U_rules(new_list, s, S)
  23: end if
  24: new_clause  $\leftarrow$  case_analysis(clause_list, s, S) {see if we can generate a clause for case
    analysis; this is obtained by unifying a literal of an input clause with the complement of
    an eligible literal}
  25: if new_clause  $\neq$  fail then
  26:   let new_list be clause_list with new_clause added to the end
  27:   return try_U_rules(new_list, s, S)
  28: end if
  29: return "fail"
end try_U_rules

```

clauses describing axioms in set theory, and 2 hypothesis clauses and 1 conjecture clause, which express the negation of the theorem. The predicates are member for \in , subset for \subset , and equal_sets for set equality. member_of_1_not_of_2 is a Skolem function that takes 2 arguments. Semantically, each of the arguments represents a set and the function maps to an element that is in the set represented by the first argument but not in the set represented by the second argument. element_of_b and b are constants. Subset, Superset and Element are variables. Axiom 1 expresses membership in subsets and states that $\forall \text{Element } \forall \text{Subset } \forall \text{Superset } (\text{Subset} \subset \text{Superset} \wedge \text{Element} \in \text{Subset}) \Rightarrow \text{Element} \in \text{Superset}$, i.e., an element of a set is also an element of a superset of the set. Axioms 2 and 3 express set membership in terms of subset and state that $\exists \text{Element } \forall \text{Subset } \forall \text{Superset } (\text{Element} \in \text{Superset} \wedge \text{Element} \notin \text{Subset}) \Rightarrow \text{Subset} \subset \text{Superset}$. Axioms 4-6 express equality in terms of subset. Axioms 4 and 5 express that two sets being equal implies each is a subset of the other; Axiom 6 expresses that two sets that are subsets of each other are equal sets. Hypothesis 1 and 2 state that b and bb are equal sets, and that element_of_b is an element of the set b. Then the conjecture to prove is that element_of_b is an element of the set bb and the negation of this conjecture is stated by the conjecture clause.

```
% membership_in_subsets, axiom 1.
```

```
{not(member(Element,Subset)),
not(subset(Subset,Superset)),
member(Element,Superset)}.
```

```
% subsets_axiom1, axiom 2.
```

```
{subset(Subset,Superset),
member(member_of_1_not_of_2(Subset,Superset),Subset)}.
```

```
% subsets_axiom2, axiom 3.
```

```
{not(member(member_of_1_not_of_2(Subset,Superset),Superset)),
subset(Subset,Superset)}.
```

```
% set_equal_sets_are_subsets1, axiom 4.
```

```
{not(equal_sets(Subset,Superset)),
subset(Subset,Superset)}.
```

```
% set_equal_sets_are_subsets2, axiom 5.
```

```
{not(equal_sets(Superset,Subset)),
subset(Subset,Superset)}.
```



```

% subsets_are_set_equal_sets, axiom 6.
{not(subset(Set1,Set2)),
 not(subset(Set2,Set1)),
 equal_sets(Set2,Set1)}.

% b_equals_bb, hypothesis 1.
{equal_sets(b,bb)}.

% element_of_b, hypothesis 2.
{member(element_of_b,b)}.

% prove_element_of_bb, conjecture.
{not(member(element_of_b,bb))}.

```

Initially, the ascending sequence, S_0 , is empty. The application of a rule of inference, R_i , to ascending sequence S_{i-1} results in the new ascending sequence S_i . Shown below are the successive ascending sequences as well as the rules applied leading up to the ascending sequence S_9 , which contains the empty clause $\{\}$.

$S_0 : []$

$S_1 : [\{ \text{not}(\text{equal_sets}(b,bb)), \text{subset}(b,bb) \}]$ Applying **Extension** to S_0 .

$S_2 : [\{ \text{not}(\text{equal_sets}(b,bb)), \text{subset}(b,bb) \}, \{ \text{member}(\text{element_of_b}, \text{Superset}), \text{not}(\text{member}(\text{element_of_b}, b)), \text{not}(\text{subset}(b, \text{Superset})) \}]$ Applying **Extension** to S_1 .

$S_3 : [\{ \text{not}(\text{equal_sets}(b,bb)), \text{not}(\text{member}(\text{element_of_b}, b)), \text{member}(\text{element_of_b}, bb) \}]$ Applying **Resolution** to S_2 .

$S_4 : [\{ \text{not}(\text{equal_sets}(b,bb)), \text{not}(\text{member}(\text{element_of_b}, b)), \text{member}(\text{element_of_b}, bb) \}, \{ \text{not}(\text{member}(\text{element_of_b}, bb)) \}]$ Applying **Extension** to S_3 .

$S_5 : [\{ \text{not}(\text{equal_sets}(b,bb)), \text{not}(\text{member}(\text{element_of_b}, b)) \}]$ Applying **Resolution** to S_4 .

$S_6 : [\{ \text{not}(\text{equal_sets}(b,bb)), \text{not}(\text{member}(\text{element_of_b}, b)) \}, \{ \text{member}(\text{element_of_b}, b) \}]$ Applying **Extension** to S_5 .

$S_7 : [\{ \text{not}(\text{equal_sets}(b,bb)) \}]$ Applying **Resolution** to S_6 .

$S_8 : [\{ \text{not}(\text{equal_sets}(b,bb)) \}, \{ \text{equal_sets}(b,bb) \}]$ Applying **Extension** to S_7 .

$S_9 : [\{ \}]$ Applying **Resolution** to S_8 .

In the above example, the proof was found entirely by the application of basic rules. The next example illustrates the operation of U rules; the empty clause is derived by application of U rules as well as basic rules. The theorem being proved is that the union of a set, a , with itself is the set, a , itself ($a = a \cup a$). Following are the 10 input clauses — 9 axioms (6 of them being identical to the ones in the previous example, the other 3 being axioms defining set union), and the conjecture clause, which is the negation of the theorem. The predicates *member*, *subset*, *equal_sets* and *union* have been replaced by the symbols \in , \subset , $=$, and \cup , respectively, for better readability. Axioms 1-6 are the same as in the preceding example. Axioms 7-9 express set union in terms of set membership. Axiom 7 states that $\forall A \forall B \forall C (A \notin B \wedge A \notin C) \Rightarrow (A \notin B \cup C)$, i.e., if an element is a member of neither of two sets then the element is not a member of the union of the two sets. Axioms 8 and 9 state that $\forall A \forall B \forall C A \in B \Rightarrow A \in B \cup C$, and $\forall A \forall B \forall C A \in B \Rightarrow A \in C \cup B$, respectively, i.e., an element being a member of a set implies that the element is also a member of the union of the set with another set, where the union operation is commutative. The conjecture clause states the negation of the theorem to be proved.

% membership_in_subsets, axiom 1.
 $\{ \text{not}(X \in Y), \text{not}(Y \subset Z), X \in Z \}.$

% subsets_axiom1, axiom 2.
 $\{ Y \subset Z, g(Y, Z) \in Y \}.$

% subsets_axiom2, axiom 3.
 $\{ \text{not}(g(Y, Z) \in Z), Y \subset Z \}.$

% set_equal_sets_are_subsets1, axiom 4.
 $\{ \text{not}(Y = Z), Y \subset Z \}.$

% set_equal_sets_are_subsets2, axiom 5.
 $\{ \text{not}(Z = Y), Y \subset Z \}.$

% subsets_are_set_equal_sets, axiom 6.

{not($X \subset Y$), not($Y \subset X$), $Y = X$ }.

% union_definition_1, axiom 7.

{not($A \in B \cup C$), $A \in B$, $A \in C$ }.

% union_definition_2, axiom 8.

{not($A \in B$), $A \in B \cup C$ }.

% union_definition_3, axiom 9.

{not($A \in B$), $A \in C \cup B$ }.

% conjecture.

{not($a = a \cup a$)}.

Following are the successive ascending sequences resulting from the application of the OSHL-U rules. The input clause on which a U rule is applied is specified when a U clause is added to the ascending sequence. Also, the eligible literal in a U clause in the ascending sequence is highlighted.

$S_0 : []$

$S_1 : [\{ \text{not}(a = a \cup a) \}]$ (Extension, conjecture clause)

$S_2 : [\{ \text{not}(a = a \cup a) \}, \{ \text{not}(a \cup a \subset a) \}, \text{not}(a \subset a \cup a), a = a \cup a]$ (Case Analysis, axiom 6)

$S_3 : [\{ \text{not}(a = a \cup a) \}, \{ \text{not}(a \cup a \subset a) \}, \text{not}(a \subset a \cup a), a = a \cup a, \{ \text{not}(g(a \cup a, a) \in a), a \cup a \subset a \}]$ (UR Resolution, axiom 3)

$S_4 : [\{ \text{not}(a = a \cup a) \}, \{ \text{not}(a \cup a \subset a) \}, \text{not}(a \subset a \cup a), a = a \cup a, \{ \text{not}(g(a \cup a, a) \in a), a \cup a \subset a \}, \{ a \cup a \subset a, g(a \cup a, a) \in a \cup a \}]$ (UR Resolution, axiom 2)

$S_5 : [\{ \text{not}(a = a \cup a) \}, \{ \text{not}(a \cup a \subset a) \}, \text{not}(a \subset a \cup a), a = a \cup a, \{ \text{not}(g(a \cup a, a) \in a), a \cup a \subset a \}, \{ a \cup a \subset a, g(a \cup a, a) \in a \cup a \}, \{ \text{not}(g(a \cup a, a) \in a \cup a), g(a \cup a, a) \in a \}]$ (Unit Filter, axiom 7)

$S_6 : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \cup \mathbf{a} \subset \mathbf{a}), \text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\text{not}(\mathbf{g}(\mathbf{a} \cup \mathbf{a}, \mathbf{a}) \in \mathbf{a}), \mathbf{a} \cup \mathbf{a} \subset \mathbf{a}\}, \{\mathbf{a} \cup \mathbf{a} \subset \mathbf{a}, \mathbf{g}(\mathbf{a} \cup \mathbf{a}, \mathbf{a}) \in \mathbf{a}\}]$ (U Resolution)

$S_7 : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \cup \mathbf{a} \subset \mathbf{a}), \text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\mathbf{a} \cup \mathbf{a} \subset \mathbf{a}\}]$ (U Resolution)

$S_8 : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}]$ (U Resolution)

$S_9 : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}, \mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a}\}]$ (UR Resolution, axiom 2)

$S_{10} : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}, \mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a}\}, \{\text{not}(\mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a} \cup \mathbf{a}), \mathbf{a} \subset \mathbf{a} \cup \mathbf{a}\}]$ (UR Resolution, axiom 3)

$S_{11} : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}, \mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a}\}, \{\text{not}(\mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a} \cup \mathbf{a}), \mathbf{a} \subset \mathbf{a} \cup \mathbf{a}\}, \{\text{not}(\mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a}), \mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a} \cup \mathbf{a}\}]$ (Unit Filter, axiom 7)

$S_{12} : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}, \mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a}\}, \{\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}, \text{not}(\mathbf{g}(\mathbf{a}, \mathbf{a} \cup \mathbf{a}) \in \mathbf{a})\}]$ (U Resolution)

$S_{13} : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}), \mathbf{a} = \mathbf{a} \cup \mathbf{a}\}, \{\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}\}]$ (U Resolution)

$S_{14} : [\{\text{not}(\mathbf{a} = \mathbf{a} \cup \mathbf{a})\}, \{\mathbf{a} = \mathbf{a} \cup \mathbf{a}\}]$ (U Resolution)

$S_{15} : [\{\}]$ (Resolution)

The initial interpretation used is a trivial all-positive interpretation, that is, every predicate is interpreted to “true”. Initially, the ascending sequence is empty. The conjecture clause is added to the sequence by Extension. Then Case Analysis is applied on an instance of axiom 6, obtained by using the substitution $\{X \rightarrow \mathbf{a} \cup \mathbf{a}, Y \rightarrow \mathbf{a}\}$. Each of the literals, $\text{not}(\mathbf{a} \cup \mathbf{a} \subset \mathbf{a})$, $\text{not}(\mathbf{a} \subset \mathbf{a} \cup \mathbf{a})$, is removed by applying a sequence of UR Resolution and Unit Filter; the remaining literal is removed by Resolution with the conjecture clause, generating the empty clause. Note that the two cases being removed with U rules correspond to proving the subgoals, $\mathbf{a} \cup \mathbf{a} \subset \mathbf{a}$ and $\mathbf{a} \subset \mathbf{a} \cup \mathbf{a}$.

3.4 Heuristics for Proof Search

The OSHL-U search for instances is guided by additional heuristics that are not used in OSHL. The heuristics are designed to provide better control over which instances are generated. These are described now.

3.4.1 Delta Size Measure for Clause Instances

Selection of a minimal instance that will subsequently be used to refine the current interpretation is a crucial part of the OSHL strategy. However, because a syntactic measure is used to compute the order of instances, it is possible for the selection to become biased in favor of syntactically small input clauses. The delta size measure used by OSHL-U for computing order of clause instances tries to overcome this bias.

OSHL uses a size-lexical order on clauses, when looking for the minimal clause instance. The size of a clause is the size of the largest literal in the clause. The size of a literal is computed using the number of symbols occurring in the literal. A symbol is a variable, constant, function name or predicate name. A symbol contributes a count of 1 to the size of a literal for every occurrence of the symbol in the literal. The sum of the size measures contributed by all the symbols occurring in a literal is the size of the literal. So, the literals $L_1 = at(City, Leaves, Time, Situation)$ and $L_2 = at(City, Leaves, s(Time), wait_at(Situation))$ have sizes of 5 and 7, respectively, and the clause $C = \{L_1, L_2\}$ has a size of 7, which is the maximum of the sizes of the literals in C .

In the OSHL algorithm, a ground clause is generated by instantiating variables in an input clause. Therefore, size of a ground clause depends on the size of the uninstantiated input clause as well as the size of ground terms used to instantiate variables. This biases the instance selection in favor of input clauses that are smaller to begin with. For example, the minimum size for L_2 is 7, while that for L_1 is 5. So no matter how L_2 is instantiated, its size as computed by OSHL will be greater than all instances of L_1 of sizes 5 and 6. Instance selection, therefore, is biased in favor of L_1 over L_2 .

OSHL-U does not use the absolute symbol size measure that OSHL uses for ground clauses. Instead, OSHL-U uses a relative measure, the increment in the size of a clause due to instantiation. This measure is referred to as the *delta size* of the ground clause. If the symbol size measure of an uninstantiated clause C is s_C and the symbol size of an instance $C\theta$ is $s_{C\theta}$, then the delta size measure for $C\theta$ is $(s_{C\theta} - s_C)$. Note that a ground clause generated by OSHL-U is an instance of an input clause, so it is possible to compute the delta size for every ground clause.

The delta size used by OSHL-U for minimal instance selection is not affected by the original sizes of the uninstantiated input clauses. This is helpful in that it does not bias the minimal instance selection in favor of ground clauses which are instances of small input clauses. Otherwise, many instances of a small input clause may be generated before generating any instance of a larger input clause.

3.4.2 Favoring Ground Terms in Generating Minimal Instances

The measure of syntactic size for terms in OSHL-U favors ground terms occurring in the non-axiom input clauses, that is, in the hypotheses and conjecture clauses. This is different than in OSHL. In OSHL, the size of a term is simply the number of symbol occurrences. Each symbol contributes 1 to the syntactic size of a term for each time it occurs in the term. OSHL-U regards the syntactic size of a ground term occurring in a non-axiom input clause as 1. In other words, such a term is treated like a single symbol. For example, in the commutativity theorem for set union, the terms *union*(*a*, *b*) and *union*(*b*, *a*) occurring in the conjecture clause are ground terms; in OSHL-U each of these terms is considered to be of syntactic size 1, though the size of each would be 3 in OSHL. In this way, OSHL-U tries to ensure that bigger ground terms that occur in the theorem do not bias the minimal clause selection against selection of instances containing such terms. Otherwise, it is possible that many syntactically smaller clauses not containing any of these terms would be generated before larger clauses containing these terms. Being part of the theorem, these terms are likely to be essential to the proof. By generating clauses containing such terms earlier, OSHL-U avoids having to search first through many smaller clauses that do not contain such terms.

3.4.3 Relevance Distance from Input Clauses

Another heuristic used in minimal ground instance generation in the OSHL-U implementation is how closely a clause is related to some input clause. We call this the *relevance* of a clause to the set of input clauses. The OSHL algorithm generates clauses with smaller absolute size before those with larger. So OSHL searches proofs with short clauses before proofs with longer clauses. This favors short clauses in the proof search even if a longer clause could be linked more closely to an input clause. The idea behind using relevance in OSHL-U is to consider instances more closely related to input clauses before considering instances that are further away. Relevance allows the generation of longer instances that are linked closely to the input clauses before shorter instances that are not as closely linked.

Relevance is estimated as follows. Each fully instantiated clause is associated with a non-negative integer representing its relevance to the set of input clauses. This number is called the *relevance distance*. A smaller relevance distance indicates that an instance is more closely related to the set of input clauses. In selecting instances, the sum of the relevance distance

and the delta size is considered and a smaller sum is favored. So when two instances have the same delta size, the one with a smaller relevance number is favored. Note also that it is possible to generate an instance with a larger delta size before one with a smaller delta size, if the former is more closely related to the set of input clauses. Note that relevance distance does not need to be computed for clauses which are not ground

Relevance distances are computed in the following way. Every input clause is assigned a relevance distance of zero, though these could also be assigned other values by the user, if desired. Also, every eligible literal under the initial interpretation is assigned a relevance distance of zero. An instance of an input clause, none of whose literals are complements of eligible literals, has the same relevance distance as the input clause itself, that is, zero. The relevance distance of an eligible literal is the same as the relevance distance of the clause from which it is selected. A literal that is a complement of an eligible literal is assigned the relevance distance of that eligible literal incremented by 1. The relevance distance of a clause, several of whose literals are complements of eligible literals, is the smallest of the relevance distances of all such literals.

3.5 Implementation

OSHL-U is implemented in Ocaml on Linux. The actual implementation of the rules in OSHL-U has additional restrictions; for example, the clause instance $C\theta$ used for extension must be a minimal clause contradicting a minimal interpretation of E where clauses and interpretations are ordered in a specified way. The semantics provide the initial interpretation. The current interpretation is a refinement of the initial interpretation. A selected instance is always falsified by the current interpretation. The choice of semantics, therefore, influences the selection of the minimal instance. We describe the implementation of the OSHL algorithm for instantiating instances — the G_S^{min} algorithm for computing the minimal ground instance — which implements the rule of extension. It requires *disunification* and *eligibility substitution*, which we will define and discuss next. For a more detailed treatment of the OSHL core algorithm, see [PZ00]. We also discuss how the semantics are used in computing the minimal instance.

3.5.1 Disunification

Suppose that M is a ground literal and L is an arbitrary literal having M as an instance. Define $M//L$, the *prefixes of M relative to L* , to be the smallest set of literals such that the following conditions are met:

- $L \in M//L$

- if $L' \in M//L$ and x is the leftmost variable in L' , then $L'\{x \leftarrow f(x_1, \dots, x_n)\} \in M//L$, where x_1, \dots, x_n are distinct variables that do not appear in L' , and where f is chosen so that M is an instance of $L'\{x \leftarrow f(x_1, \dots, x_n)\}$.

For example, if L is $P(x, f(y))$ and M is $P(a, f(f(b)))$ and the set of function (a constant is a function that takes no arguments) symbols is $\{a, b, f\}$, then $M//L$ is $\{P(x, f(y)), P(a, f(y)), P(a, f(f(y))), P(a, f(f(b)))\}$.

Suppose that M is a ground literal and L is an arbitrary literal having M as an instance. Then $\text{dis2}(L, M)$ is the set of literals of the form $L'\{x \leftarrow f(x_1, \dots, x_n)\}$ such that $L' \in M//L$, L' is not ground, x is the leftmost variable of L' , x_1, \dots, x_n are distinct variables that do not appear in L' , and such that f is chosen so that M is *not* an instance of $L'\{x \leftarrow f(x_1, \dots, x_n)\}$. For L and M in the above example, $\text{dis2}(L, M)$ is $\{P(a, f(a)), P(a, f(b)), P(a, f(f(a))), P(a, f(f(f(x_1))))\}$.

Note that no two distinct elements of $\text{dis2}(L, M)$ unify with each other. The number of elements in $\text{dis2}(L, M)$ is polynomial in $\|L\| + \|M\|$ and the symbol sizes of these elements are also polynomial in $\|L\| + \|M\|$. The algorithm $\text{dis}(L, \mathcal{L})$ given below computes disunification of a literal L and a set of ground literals \mathcal{L} , that is, it creates a list S of instances of L such that an instance L' of L is not in the set \mathcal{L} of ground literals iff L' is an instance of some element of S . While dis2 computes the disunification of a literal with a ground literal, dis computes the disunification of a literal with a set of ground literals. So dis2 serves as a helper procedure to dis . To instantiate a literal of a clause in OSHL, we need the disunification of the literal with the set of eligible literals. That is done by the procedure dis shown in Algorithm 5 where L is the literal and \mathcal{L} is the set of ground literals.

Algorithm 5 Disunification of a literal L with a set of ground literals \mathcal{L}

procedure $\text{dis}(L, \mathcal{L})$

```

1: if  $\mathcal{L}$  is empty then
2:   return  $\{L\}$ 
3: else
4:   let  $L_1$  be an element of  $\mathcal{L}$ 
5:   if  $L$  and  $L_1$  do not unify then
6:     return  $\text{dis}(L, \mathcal{L} - \{L_1\})$ 
7:   else if  $L \equiv L_1$  then
8:     return  $\{\}$ 
9:   else
10:    return  $(\bigcup \{\text{dis}(M, \mathcal{L} - \{L_1\}) : M \in \text{dis2}(L, L_1)\})$ 
11:  end if
12: end if
end dis

```

If the set of ground literals \mathcal{L} is empty (line 1), then there are no literals to disunify with and the procedure returns the set containing the literal L . Otherwise, the procedure considers an element L_1 of \mathcal{L} (line 4). If L_1 does not unify with L (line 5), then the procedure recurses on the remaining set of ground literals $\mathcal{L} - \{L_1\}$ (line 6). Otherwise, if L and L_1 are equivalent (line 7), i.e., L is equivalent to an eligible literal already, then L cannot be disunified with the set of eligible literals so the empty set is returned 8. Otherwise, the disunifier of the literal L with the literal L_1 , $\text{dis2}(L, L_1)$, is computed; then each element of $\text{dis2}(L, L_1)$ is disunified recursively with the remaining eligible literals $\mathcal{L} - \{L_1\}$ and the union of all these disunification sets is returned (line 10).

In most cases, L and L_1 will not unify, and the size of the set returned will be small. On recursive calls of dis , the literal L becomes larger, making unification even less likely. So the procedure is efficient in practice. As will be seen in the G_S^{min} algorithm, the search for instances is bounded by size, i.e., we are looking for instances of a certain size s ; this means that if $\|L\| > s$ then $\text{dis}(L, \mathcal{L})$ can return the empty set. This also makes the procedure efficient. Also, all the elements of dis are not computed at once, but they can be computed one by one, stopping as soon as an instance has been found by G_S^{min} .

3.5.2 Eligibility Substitution

An *eligibility substitution* for a clause C and a set $\{L_1, \dots, L_n\}$ of eligible literals is a most general substitution α such that for every literal L in C , $L\alpha$ is an instance of some literal in $\text{dis}(L, \{L_1, \dots, L_n\}) \cup \{\neg L_1, \dots, \neg L_n\}$.

Ground instances can be obtained by first instantiating clauses of S , the set of input clauses, with eligibility substitutions and then instantiating to a ground clause that contradicts the current interpretation. Suppose the current interpretation is $I_0[L_1 \dots L_n]$. Recall that this means the initial interpretation was I_0 and the current interpretation is identical to I_0 except that it satisfies the eligible literals L_1, \dots, L_n . Then we try to unify each literal L of an input clause C with an element of $\text{dis}(L, \{L_1, \dots, L_n\}) \cup \{\neg L_1, \dots, \neg L_n\}$ and then further instantiate the literals in $C\alpha - \{\neg L_1, \dots, \neg L_n\}$ so that they contradict the initial interpretation, I_0 . The eligibility substitution α replaces some or all variables of C by ground terms making the latter instantiation easier. For a fixed S , the number of literals in C is bounded, so computing eligibility substitution can be done in polynomial time.

3.5.3 The Instantiation Algorithm

The procedure G_S^{min} shown in Algorithm 6 computes the set U of all minimal terms of increasing sizes. The set U is used to compute a set V of instances in S which contradict current interpretation I , and a minimal element of V is returned. If there are a finite number

of terms and no instance of S contradicting I is found, then G_S^{min} halts and returns “fail”. But if there are infinitely many terms and no instance of S contradicting I is found, then G_S^{min} will not terminate. $>_{lex}$ is the lexicographic ordering.

Algorithm 6 The $G_S^{min}[I]$ procedure to compute a ground instance minimal in $>_{lex}$ that contradicts interpretation I

```

procedure  $G_S^{min}[I]$ 
1: let  $\{L_1, \dots, L_n\}$  be such that  $I$  is  $I_0[L_1 \dots L_n]$ 
2:  $\{ \text{Compute minimal terms relative to } I \text{ and } >_{lex} \}$ 
3:  $U \leftarrow \{ \}$   $\{U \text{ is the set of minimal terms}\}$ 
4:  $s = 0$   $\{\text{Initialize size of instance to } 0\}$ 
5: repeat
6:    $U_{new} \leftarrow \{ \}$   $\{U_{new} \text{ is the set of minimal terms of size } s\}$ 
7:    $U_1 \leftarrow \{ \text{terms } f(s_1, \dots, s_k) \text{ of size } s : s_i \in U, f \in \mathcal{F} \}$ 
8:   for all terms  $t$  in  $U_1$  in order of  $>_{lex}$  do
9:     if there does not exist  $u$  in  $U \cup U_{new}$  such that  $t^{I_0} = u^{I_0}$  then
10:       $U_{new} \leftarrow U_{new} \cup \{t\}$ 
11:    end if
12:  end for
13:  if  $U_{new}$  is empty then
14:    return “fail”
15:  end if
16:   $U \leftarrow U \cup U_{new}$ 
17:   $V \leftarrow \{ \}$   $//V \text{ is a set of instances of } S$ 
18:  for  $C \in S$  do
19:    for all eligibility substitutions  $\alpha$  of  $C$  do
20:       $D \leftarrow C\alpha - \{ \neg L_1, \dots, \neg L_n \}$ 
21:      for all  $\beta$  of the form  $\{x_1 \leftarrow t_1, \dots, x_m \leftarrow t_m\}$  where  $x_1, \dots, x_m$  are the variables
        in  $D$  and  $t_1, \dots, t_m$  are in  $U$  do
22:        if  $I_0 \not\models D\beta$  then
23:           $V \leftarrow V \cup \{C\alpha\beta\}$ 
24:        end if
25:      end for
26:    end for
27:  end for
28:   $s \leftarrow s + 1$   $\{\text{Increment size of ground instance for next iteration}\}$ 
29: until  $V$  is not empty
30: return an element of  $V$  that is minimal in the clause ordering
end  $G_S^{min}$ 

```

Given the set of input clauses S , the G_S^{min} procedure computes a $>_{lex}$ minimal instance that contradicts a given interpretation I as follows. Suppose L_1, L_2, \dots, L_n are the eligible literals (line 1). U , the set of minimal terms, is initialized to the empty set (line 3). Instances are generated in order of increasing size so the size, s , is initialized to 0 (line 4). The loop in

lines 5 - 29 is repeated till an instance is found. U_{new} maintains the set of minimal terms of the current size s , and it is initialized to be empty (line 6). All terms of size s that are functions or symbols are stored in U_1 (line 7). For every term in U_1 , where terms are considered in increasing order of $>_{lex}$ (line 8), if there is not already a term in U or U_{new} that is equivalent to the term under the initial interpretation (line 9), then the term is added to the set of terms U_{new} of size s (line 10). If there are no terms of the current size, i.e., U_{new} is empty (line 13), then the G_S^{min} procedure fails (line 14); otherwise all the terms in U_{new} are added to the set of minimal terms U (line 16). V maintains a set of instances of the input clauses S that are False under the current interpretation and is initialized to be empty (line 17). For every clause C in the input set S , every eligibility substitution α is considered in turn — if D is the clause resulting from applying eligibility substitution α to C and removing any literals that are negated eligible literals (line 20), then every possible way of substituting the variables in D with a minimal term from U is looked at (line 21). If any such substitution gives a clause that is False under the initial interpretation (line 22), then an instance that is False under the current interpretation is obtained by applying this substitution to the eligible substitution of C (line 23). To see why this instance, denoted by $C\alpha\beta$ on line 23, is False under the current interpretation, consider that this instance, with any instance of an eligible literal removed from it, is False under the initial interpretation. The current interpretation differs from the initial interpretation only in the set of eligible literals, the eligible literals being True in the current interpretation. A negated eligible literal is False under the current interpretation, so the instance $C\alpha\beta$ is False under the current interpretation. The instance is added to the set, V , of instances that are False under the current interpretation. If no instance can be added to V , i.e., V is empty, then the loop (line 5 - line 29) is repeated for a larger size of instances (line 28), till some instance can be added to V . The instance finally returned by the procedure is that which is minimal in the clause ordering among all the instances in V (line 30).

In the earlier Prolog implementation of OSHL, G_S^{min} was implemented differently. Some of the literals of an input clause C were unified with complements of eligible literals and the remaining variables of C were replaced with minimal or non-minimal terms in such a way that no new eligible literals are created and so that the resulting clause contradicts I .

3.5.4 How Semantics Are Used

Semantics are used in the test for $I_0 \not\models D\beta$ in the G_S^{min} procedure. The OSHL-U implementation allows the user to specify an all-positive or an all-negative initial semantics, which we refer to as trivial semantics, or to specify a non-trivial semantics. The trivial semantics are provided by the system. To specify a non-trivial semantics, the user writes Ocaml functions that define the domain elements and evaluate the functions and predicates in the problem. If the user specifies only a partial semantics, the functions and predicates not specified by the

user will be mapped, by default, to those of one of the trivial semantics. When a user-specified semantics is supplied, then it will be used to evaluate the truth value of the ground clause $D\beta$.

3.5.5 Proofchecker

Once a proof is detected by the empty clause being added to the ascending sequence, the proof is reconstructed from the stored clauses by tracing back through the sequence of rule applications. The proof of the empty clause thus generated is printed in a form that is easy to read by a human. Each statement in the proof is either an instance of an input clause or the result of resolution of two statements. In addition, the implementation includes a proofchecker which verifies the soundness of a proof found by OSHL-U by checking for the soundness of each inference step in the proof. Following is an example of a proof generated by OSHL-U for problem SET001-1 from the SET (set theory) domain of the TPTP. The problem and the derivation of the proof was given in the first example of OSHL-U operation. Each line in the proof consists of a clause which is either an instance of an input clause or obtained by resolving two other clauses, each of which is indented one tab beyond the current clause. In this example, the empty clause on line 1 is obtained by resolving an instance of input clause (7) shown on line 2 and the clause shown on line 3. The clause on line 3, in turn, is obtained by resolving the two clauses on lines 4 and 5, the former being an instance of input clause 8 and the latter being the result of resolving the two clauses on lines 6 and 7. The clause on line 6 is an instance of the input clause 9, while the clause on line 7 is the result of resolving the two clauses on lines 8 and 9, which are instances of input clauses 1 and 4, respectively. Thus, the proof gives a chain of inference that derives the empty clause from a set of instances of the input clauses.

```

1. { }
2.   input clause (7) { equal_sets( b bb) }
3.   { not( equal_sets( b bb) ) }
4.     input clause (8) { member( element_of_b b) }
5.     { not( equal_sets( b bb) ), not( member( element_of_b b) ) }
6.       input clause (9) { not( member( element_of_b bb) ) }
7.       { not( equal_sets( b bb) ), not( member( element_of_b b) ),
          member( element_of_b bb) }
8.         input clause (1) { member( element_of_b Superset),
                             not( member( element_of_b b) ), not( subset( b Superset) ) }
9.         input clause (4) { not( equal_sets( b bb) ),
                             subset( b bb) }

```


Chapter 4

Experimental Evaluation of OSHL-U Efficiency

The efficiency of OSHL-U was measured experimentally. The execution times were compared to those obtained with a prior implementation of OSHL, as well as to those obtained with a resolution prover, Otter. Also, the improvement in performance due to the unit rules was studied by running OSHL-U both with and without the unit rules.

4.1 Execution Time as Measure of Efficiency

People are interested in obtaining proofs using automated theorem provers quickly, or at least within a reasonable amount of time. Therefore, execution time is an important measure of an automated theorem prover's efficiency. Of course, a reasonable amount of time could vary from a few minutes to several months depending on the kind of problem, or even many years for open problems of a profound nature.

A common way of measuring theorem prover efficiency, as done in the CASC competitions [SS98b] is to set a limit on the maximum CPU time a theorem prover is allowed to execute on each problem and then determine how many proofs are obtained by the theorem prover. The experimental results in this chapter were obtained in a similar way, using 30 seconds as the CPU time limit on each problem. The justification of selecting a time limit of 30 seconds is that this makes it practical to obtain results on the set of TPTP problems within a few days. We also repeated the runs on a smaller set of problems with a maximum time limit of 5 minutes on each problem.

The set of problems used to conduct these tests are from the TPTP version 2.5.0 [SS98a]. The problems are categorized into 30 domains and statistics are provided on the difficulty ratings for each problem. The difficulty rating is represented by a number lying between 0 and

1. A lower rating indicates an easier problem, based on the performances of all the provers that have been tested on the TPTP. Because many of the provers that have been tested on these problems are resolution-based provers, the ratings are a good reflection of how hard a problem is for resolution-based strategies.

4.2 Experimental Results

The results described here were obtained with a restricted implementation of the unit filtering rule, in that the unit clauses used in unit resolution were propositional. The unit filtering rule provides a way of interfacing OSHL-U with a conventional resolution prover, because resolvents from the conventional prover can be added to the set of clauses used for filtering. But we have not conducted any experiments to test this capability.

The data presented was obtained by executing OSHL-U on the TPTP v2.5.0 problems, using trivial semantics, i.e., either an initial interpretation that interpretes all predicates to positive (all-positive) or an initial interpretation that interpretes all predicates to negative (all-negative). Therefore, OSHL-U was running as a purely syntactic prover, without semantic guidance. Otter [McC94], an efficient resolution prover, was run on the TPTP v2.5.0 problems in the autonomous mode with a CPU time limit of 30 seconds on each problem.

4.2.1 Execution Time

The execution times reported are on a Pentium 4 1.7 GHz processor with 256 MB of memory running Linux. Execution times on individual problems that were proved by both OSHL-U and Otter are shown in the scatter plot of Figure 4.1. Summary statistics of the number of proofs obtained by OSHL-U and Otter is shown in Table 4.1 and Table 4.2. Table 4.1 compares the number of proofs obtained by OSHL-U and Otter breaking up the problem sets into problems of TPTP rating zero and problems of TPTP rating greater than zero. Table 4.2 compares the number of proofs obtained by OSHL-U and Otter based on the problems being Horn or non-Horn, and further based on TPTP rating within the non-Horn category.

Since OSHL-U is not designed to detect satisfiability, we collected the results on only the problems which are not known to be satisfiable. Out of 4417 such problems, OSHL-U obtained 900 proofs and Otter obtained 1697 proofs.

Comparing the total number of proofs Otter and OSHL-U obtained, we note that the number of problems OSHL-U proves is more than half the number of problems that Otter proves. This is interesting, especially considering that OSHL-U has no special rules for equality, does

Dom.	#Prob.	#Otter Proofs			#OSHL-U Proofs		
		Total	Rating=0	Rating>0	Total	Rating=0	Rating>0
ALG	9	1	1	0	0	0	0
ANA	10	0	0	0	0	0	0
BOO	64	8	6	2	0	0	0
CAT	52	25	18	7	8	7	1
COL	150	59	54	5	2	1	1
COM	9	6	6	0	3	3	0
FLD	143	28	17	11	42	18	24
GEO	187	99	50	49	64	28	36
GRA	1	1	1	0	1	1	0
GRP	503	112	105	7	52	36	16
HEN	64	43	18	25	11	11	0
HWC	4	1	0	1	0	0	0
HWV	73	48	33	15	23	18	5
KRS	9	9	9	0	9	8	1
LAT	48	6	5	1	1	1	0
LCL	422	131	103	28	15	15	0
LDA	23	0	0	0	0	0	0
MGT	134	86	68	18	46	30	16
MSC	11	9	7	2	4	2	2
NLP	43	33	29	4	4	4	0
NUM	63	18	13	5	11	10	1
PLA	30	5	5	0	1	1	0
PUZ	58	46	38	8	40	28	12
RNG	82	22	6	16	5	2	3
ROB	23	1	1	0	0	0	0
SET	604	168	128	40	213	107	106
SWC	713	145	27	118	0	0	0
SWV	13	11	10	1	11	10	1
SYN	866	570	537	33	332	277	55
TOP	6	6	4	2	2	2	0
Total	4417	1697	1299	398	900	620	280

Table 4.1: Number of proofs found by Otter and OSHL-U compared by TPTP ratings of problems.

Dom.	# of Prob.	#Otter Proofs					#OSHL-U Proofs				
		Total	H	non-Horn			Total	H	non-Horn		
				Total	R=0	R>0			Total	R=0	R>0
ALG	9	1	0	1	1	0	0	0	0	0	0
ANA	10	0	0	0	0	0	0	0	0	0	0
BOO	64	8	8	0	0	0	0	0	0	0	0
CAT	52	25	20	5	5	0	8	4	4	4	0
COL	150	59	59	0	0	0	2	1	1	1	0
COM	9	6	3	3	3	0	3	2	1	1	0
FLD	143	28	0	28	17	11	42	0	42	18	24
GEO	187	99	2	97	49	48	64	1	63	27	36
GRA	1	1	0	1	1	0	1	0	1	1	0
GRP	503	112	72	40	36	4	52	22	30	30	0
HEN	64	43	43	0	0	0	11	11	0	0	0
HWC	4	1	1	0	0	0	0	0	0	0	0
HWV	73	48	0	48	33	15	23	0	23	18	5
KRS	9	9	1	8	8	0	9	1	8	8	0
LAT	48	6	6	0	0	0	1	1	0	0	0
LCL	422	131	127	4	4	0	15	11	4	4	0
LDA	23	0	0	0	0	0	0	0	0	0	0
MGT	134	86	26	60	42	18	46	4	42	28	14
MSC	11	9	1	8	6	2	4	1	3	1	2
NLP	43	33	8	25	21	4	4	0	4	4	0
NUM	63	18	7	11	9	2	11	2	9	8	1
PLA	30	5	4	1	1	0	1	0	1	1	0
PUZ	58	46	14	32	29	3	40	6	34	34	0
RNG	82	22	19	3	0	3	5	3	2	0	2
ROB	23	1	1	0	0	0	0	0	0	0	0
SET	604	168	2	166	126	40	213	2	211	114	97
SWC	713	145	0	145	27	118	0	0	0	0	0
SWV	13	11	2	9	1	8	11	2	9	9	0
SYN	866	570	338	232	205	27	332	212	120	107	13
TOP	6	6	0	6	5	1	2	0	2	2	0
Total	4417	1697	764	933	636	297	900	281	619	401	218

Table 4.2: Number of proofs found by Otter and OSHL-U compared over Horn and non-Horn problems. H denotes Horn. R denotes TPTP rating.

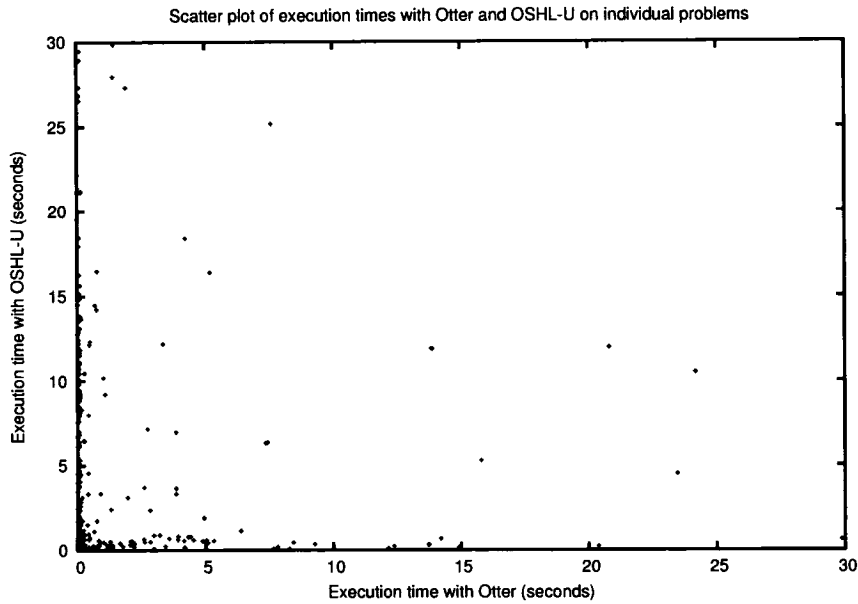


Figure 4.1: Scatter plot showing execution times with OSHL-U and Otter on individual problems that were proved by both provers.

not use term rewriting, and lacks efficient data structure support. This is the first time to our knowledge that a propositional style prover, not performing unification on non-propositional literals has demonstrated performance comparable to that of a resolution prover. In the domains of Field Theory (FLD) and Set Theory (SET), OSHL-U obtains a greater number of proofs than Otter does. So OSHL-U already performs better than Otter on these categories of problems, despite the implementation lacking a similar level of sophistication as Otter.

We looked at how many of the problems solved by the provers were Horn and non-Horn and further among the non-Horn problems, how many problems had a TPTP rating greater than 0. A higher rating indicates a harder problem, based on the performances of all the provers that were tested on the TPTP. Most of the provers tested use resolution-based strategies, so the TPTP rating is a good reflection of how hard a problem is for resolution. On non-Horn problems of rating greater than 0, OSHL-U obtains 218 proofs while Otter obtains 297 proofs. On all problems of rating higher than 0, OSHL-U obtains 280 proofs and Otter obtains 398 proofs. OSHL-U was not designed for Horn problems and does not necessarily perform better than resolution on this kind of problem. These results suggest that the performance of OSHL-U relative to resolution improves as the problems become harder for resolution and non-Horn. Thus for harder and non-Horn problems, OSHL-U could become superior to resolution.

The 30 second time limit was chosen somewhat arbitrarily to enable us to collect a set of results on all the problems within a few days. To get an idea of how the number of proofs might increase with longer execution times, the OSHL-U runs were repeated on problems from the Field Theory (FLD) domain, allowing a longer maximum time limit of 300 seconds. A total of 57 proofs were obtained. Compared to 42 proofs within the 30 second limit, this is a 35 per cent increase.

4.2.2 Performance Improvement from Unit Rules

The number of proofs obtained with the OSHL-U implementation by turning off the unit rules is shown in Table 4.3. OSHL-U when run without any of the unit rules was able to get 238 proofs compared to 900 proofs with the unit rules. This clearly indicates a performance improvement of OSHL due to the unit rules. It should be noted that OSHL was originally designed to be used only in conjunction with semantic models that provide guidance in proof search, and lacks any syntactic guidance. Considering that these tests were run with trivial semantics that do not necessarily provide useful guidance in looking for proofs, the low number of proofs obtained without the unit rules is not surprising. This also illustrates the inefficiency problem of directly enumerating the Herbrand set in searching for a proof. The performance improvement with the unit rules, seen as an increase in the number of proofs, shows that the unit rules are indeed able to provide better syntactic guidance in proof search.

4.2.3 OSHL-U Performance Compared to Other OSHL Enhancements

Table 4.4 provides summary comparisons of the results of OSHL-U to those of OSHL tested earlier with several other enhancements. Results of running OSHL with replacement rules and definition detection on 6 problems in SET were presented in [PZ99] and corresponds to the first row of results in Table 4.4.

Adnan Yahya collected the results of running OSHL with the Set Theory flag on 79 problems in SET. The Set Theory flag turned on facilities such as replacement rule with definition detection [PZ99], special rules for the equality predicate and term rewriting. This is shown in the second row in Table 4.4.

OSHL has also been used to obtain proofs with the help of semantic models on a subset of problems from the Set Theory (SET) domain [Fun01]. The three methods that were tried correspond to the last three rows in Table 4.4. The third row in Table 4.4 represents running OSHL with standard semantics, i.e., a natural semantics that models all the Set Theory axioms in the problem input. The fourth row corresponds to using OSHL with a feature called Modified Size Measure (MSM), which assigns a size of 0 to terms that are likely to be used in a proof; the modified size of 0 ensures that literals containing these terms are

Dom.	#Prob.	#Proofs with all rules	#Proofs without unit rules
ALG	9	0	0
ANA	10	0	0
BOO	64	0	0
CAT	52	8	4
COL	150	2	1
COM	9	3	0
FLD	143	42	2
GEO	187	64	1
GRA	1	1	1
GRP	503	52	7
HEN	64	11	5
HWC	4	0	0
HWV	73	23	0
KRS	9	9	5
LAT	48	1	1
LCL	422	15	11
LDA	23	0	0
MGT	134	46	8
MSC	11	4	1
NLP	43	4	0
NUM	63	11	3
PLA	30	1	0
PUZ	58	40	14
RNG	82	5	1
ROB	23	0	0
SET	604	213	27
SWC	713	0	0
SWV	13	11	4
SYN	866	332	141
TOP	6	2	1
Total	4417	900	238

Table 4.3: Number of proofs found by OSHL-U with and without U rules.

Method compared to	#Prob.	#proofs by method compared to	#proofs by OSHL-U
Oshl+repl. rules	6	6	6
Oshl+set theory flag	79	54	79
Oshl+std. semantics	88	41	88
Oshl+MSM	88	78	88
Oshl+MSM with atoms	88	83	88

Table 4.4: Number of proofs found by OSHL-U and other methods used with OSHL such as replacement rules for definition expansion, methods that work well with set theory problems (definition expansion, special rules for equality predicate, term rewriting), use of semantic guidance and Modified Size Measure (MSM) to favor specific terms and atoms in the proof.

avored in generating instances over other literals. The fifth row corresponds to using OSHL with Modified Size Measure as well as explicitly specifying atoms (i.e., literals without a sign specified) to be used and atoms to be avoided in the proof search. Use of these methods involved extensive input from the human user and, in many cases, significant help from the user in the form of specifying to the prover what terms to use and what terms to avoid in a proof. Such input from the user relies on the fact that the user already knows how to prove the problem or has an idea of what subgoals to prove in order to obtain the final proof.

These results show that on the problems tested, OSHL-U demonstrates performance that is as good as that of OSHL with replacement rule and definition detection and better than that of OSHL with equality and term rewriting, and of OSHL in conjunction with some semantic guidance. Of course, this comparison is only of a limited nature because these other methods were specific to small subsets of the TPTP and were not tested exhaustively on all the TPTP problems that OSHL-U was tested on.

4.3 Conclusions

The results in this chapter show that the use of unit rules in OSHL-U causes an improvement in performance over not using these rules. The number of proofs obtained in 30 second time limited runs on each problem increases more than four times with the unit rules. Also, the total number of proofs obtained with OSHL-U, a propositional style prover not performing unification on non-propositional literals is in the range of the number of proofs obtained with a respectable resolution prover, Otter. The performance of OSHL-U relative to Otter is better on non-Horn problems and problems with higher TPTP difficulty rating than on Horn problems and problems that are less difficult. OSHL-U already performs better than Otter on the TPTP domains of Field Theory and Set Theory which contain many problems that are highly non-Horn.

Chapter 5

Refinements to the OSHL-U Implementation

Several refinements were made to the OSHL-U implementation in order to improve its performance. Some of these increased the number of proofs in some of the TPTP domains but decreased the number of proofs in others. The total number of proofs over all the problem domains was used to determine if the refinement was effective and should be retained in the implementation or discarded. The total number of proofs obtained in 30 seconds on each problem was increased to 1,027 as a result of the refinements. Without these refinements, the number of proofs was 900. So this is about a 14% increase in the number of proofs. We also describe an attempted modification to the Unit Filtering rule that increased the number of proofs on some problem domains but decreased the number of proofs overall; this modification was, therefore, discarded.

5.1 Effective Refinements to OSHL-U Implementation

5.1.1 Avoid Repeating Computations in Unit Filter and UR

In the more general case of Unit Filter, unification of a literal in a clause with the complement of a unit clause is allowed. In this case, the literal can be viewed as one which is not unified with a literal in \overline{E} and, therefore, an instance of this literal could potentially be the result of a UR Resolution.

The implementation, however, uses filtering, which is a special case of the Unit Filter rule as described in Chapter 3. Some preliminary tests showed that Filtering performs just as well as or better than Unit Filter. Filtering does not involve unification between non-ground literals. The only cases to consider in Filtering are where D is obtainable from $C \in S$ by zero or more unit resolutions with unit clauses in S , such that these unit clauses are ground. However, we include the complements of the ground unit clauses in S among the set \overline{E} at initialization.

In implementing this rule, therefore, we only need to check if for some substitution θ of the variables in a clause $C \in S$, $C\theta \subset \overline{E}$. The UR Resolution rule is similar to the Filtering rule. In UR Resolution, we search for $C \in S$ and for a substitution θ such that for a ground literal $L \in C\theta$, $L \notin E$ and $C\theta \subset \overline{E} \cup \{L\}$. While for Filtering, we try to match every literal in a clause to a literal in \overline{E} , for UR Resolution, we try to match every but one literal in a clause to a literal in \overline{E} .

Much of the computation in Filtering is repeated in UR Resolution. The repeated computation was avoided in the OSHL-U implementation by computing and storing the literals resulting from UR Resolution when performing Filtering. This way, Filtering incurs a very small computation overhead but UR Resolution is practically free.

5.1.2 Pruning Invalid Paths from the Search

Any literal in a clause could potentially give the resulting literal from a UR Resolution, and could therefore be regarded as being skipped over. However, only one literal in the clause being considered can be skipped over. Unifying with a non-ground unit clause for Unit Filtering can be considered as skipping over a literal for UR Resolution. If a literal has already been skipped over or unified with a non-ground unit clause, then for UR Resolution to apply, no subsequent literal can be skipped over or unified with a non-ground unit clause. This observation helps to prune away useless computation paths for UR Resolution.

5.1.3 Order of Input Clauses

When performing UR resolution and filtering, the input clauses are considered longest first. This makes the computation independent of the order in which the clauses are given in the input. It turned out that this also increased the number of proofs obtained. We noted also, that considering the input clauses in the reverse order, i.e., shortest first, led to a decrease in the number of proofs.

5.1.4 Potential Unification

UR Resolution, and Unit Filtering use unification, which is computationally expensive. Before attempting unification, we can eliminate some of the eligible literals and unit clauses, which will not succeed in unifying with the literal under consideration. This is referred to as *potential unification* and it reduces the computational cost for unification.

There is a limited number of predicate symbols appearing in a problem. A literal L could potentially unify with an eligible literal or unit clause if the predicate symbol of L appears in at least one eligible literal or unit clause with the opposite sign as that in L . So, for instance,

if the predicate symbol *Subset* appears in L with a positive sign, then the only eligible literals and unit clauses that need to be considered for unification with L are those containing *Subset* with a negative sign.

Potential unification also helps to prevent some clauses from being considered at all for Unit Filtering or UR Resolution. Suppose C is an input clause. If two or more literals of C are not potentially unifiable, then C can not be used for Unit Filtering or UR Resolution. If exactly one literal of C is not potentially unifiable, then C cannot be used for Unit Filtering but it can be used for UR Resolution; in this case we know in advance which literal of C should be “skipped over”. If all literals of C are potentially unifiable, then C can be used for both UR Resolution and Unit Filtering; in this case, we have to skip over each literal in turn during UR Resolution, even if the literal is unifiable.

5.1.5 Potential Unification With Counting of Unifiers

The idea of potential unification was further refined by considering the number of potential unifiers for every literal appearing in an input clause. Unification on literals in a clause was attempted in the order of “literal with fewer potential unifiers” first.

5.1.6 Incremental Size Bound on U Rules

Computation time for applying U rules depends on the size bound on clauses used. The computation times increase exponentially with larger size bounds. If we can get a success on the U rules with a small size bound then it might be a lot faster than starting right away with the maximum size bound. So we use a small initial value for the size bound on U rules. If applying U rules fails, we try to apply the U rules again with the size bound incremented by 1, until the maximum size bound is exceeded or the U rules succeed. This is essentially the same idea as that of a depth-first search with iterative deepening.

This modification did not change the number of proofs on TPTP problems, but it was kept anyway.

5.1.7 Instantiating Negative Literals Before Positive Literals

In instantiating literals of a clause to generate a ground instance, if two literals L and M have the same number of variables, and L is positive and M is negative, then M is instantiated first. The reason is that typical interpretations make most literals false and there are only a small number of literals that are true. Thus fewer instances of M would contradict the initial interpretation.

This modification did not change the number of proofs on TPTP problems, but it was kept anyway.

5.2 Refinements That Were Discarded

5.2.1 Unit Filtering on Units Generated by Non-ground UR Resolution

We refined the Unit Filtering rule by expanding the input set with unit clauses generated from the input clauses. Let S be the set of input clauses. Then the new unit clauses are generated when the proof search starts by non-ground UR resolution on S . Let U be the set of these unit clauses. The Unit Filtering rule, as originally intended, applied only to S . But we modified it so it applies to clauses in $S \cup U$. When applying Unit Filtering to a clause $\{L_1\}$ in U , we try to find an eligible literal L_2 that unifies with $\neg L_1$; if this happens, Unit Filtering on $\{L_1\}$ succeeds.

One problem with this modified Unit Filtering is that when the input set has clauses with many literals, generating the set U could take a very long time. So the process of generating new unit clauses by UR resolution was restricted to those clauses with no more than a fixed small number of literals. We experimentally found 4 to be a good bound for most TPTP problems, so we restricted the UR resolution step to input clauses with 4 or less literals.

This modification increased the number of proofs on some domains but decreased the number of proofs overall. So it was discarded.

5.3 Results

The execution times on individual problems that were proved by both OSHL-U and Otter are shown in the scatter plot of Figure 5.1. Tables 5.2 and 5.1 show the revised statistics on numbers of proofs in different categories with the OSHL-U implementation that includes all the effective modifications described.

The effective modifications increased the total number of proofs by 14.11% from 900 before the changes to 1027 after. To analyze the increase in number of proofs by category of problems, we further looked at the percentage increase in number of proofs in the categories of Horn problems, non-Horn problems, and non-Horn problems with TPTP rating greater than 0. These are summarized in Table 5.3. As seen in these numbers, the greatest increase — 21.56% — is for the category of non-Horn problems of rating greater than 0; these are presumably hard for resolution and the kind of problems we are most interested in solving

with OSHL-U. The increase in number of proofs non-Horn problems overall is 15.67% and the increase in number of proofs on Horn problems is 10.68%.

The modifications increased the number of proofs in 10 out of 30 problem domains. The most significant increases were observed in problems from Software Computation (SWC), Field Theory (FLD) and Syntactic Category (SYN).

A decrease in the number of proofs was observed in 6 problem domains. On the remaining 14 problem domains, the modifications did not change the number of proofs.

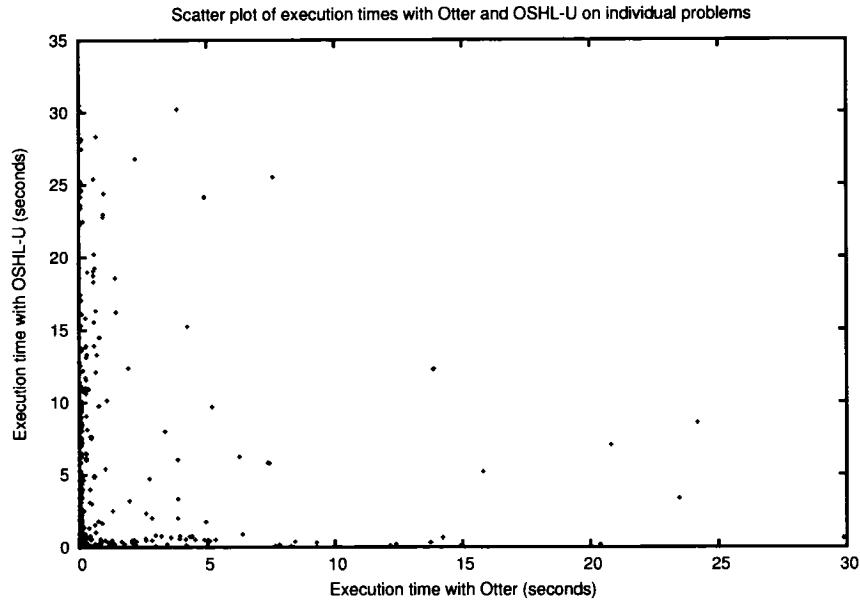


Figure 5.1: Scatter plot showing execution times with the refined implementation of OSHL-U and with Otter on individual problems that were proved by both provers.

5.4 Conclusion

The optimizations increased the number of proofs by 14.11% overall. The number of proofs increased in all categories of problems. However, the greatest improvement was in the category of non-Horn problems of rating greater than 0; here the number of proofs increased by 21.56%. So the modifications were effective in addressing the category of problems we are most interested in because these are presumably hard for resolution.

Dom.	#Prob.	#Otter Proofs			#OSHL-U Proofs		
		Total	Rating=0	Rating>0	Total	Rating=0	Rating>0
ALG	9	1	1	0	1	1	0
ANA	10	0	0	0	0	0	0
BOO	64	8	6	2	1	1	0
CAT	52	25	18	7	13	11	2
COL	150	59	54	5	1	1	0
COM	9	6	6	0	3	3	0
FLD	143	28	17	11	68	21	47
GEO	187	99	50	49	67	27	40
GRA	1	1	1	0	1	1	0
GRP	503	112	105	7	48	47	1
HEN	64	43	18	25	11	11	0
HWC	4	1	0	1	0	0	0
HWV	73	48	33	15	26	20	6
KRS	9	9	9	0	6	6	0
LAT	48	6	5	1	1	1	0
LCL	422	131	103	28	13	13	0
LDA	23	0	0	0	0	0	0
MGT	134	86	68	18	38	27	11
MSC	11	9	7	2	4	2	2
NLP	43	33	29	4	4	4	0
NUM	63	18	13	5	13	12	1
PLA	30	5	5	0	0	0	0
PUZ	58	46	38	8	40	34	6
RNG	82	22	6	16	5	2	3
ROB	23	1	1	0	0	0	0
SET	604	168	128	40	211	118	93
SWC	713	145	27	118	66	22	44
SWV	13	11	10	1	11	10	1
SYN	866	570	537	33	373	357	16
TOP	6	6	4	2	2	2	0
Total	4417	1697	1299	398	1027	754	273

Table 5.1: Number of proofs found by Otter and OSHL-U compared by TPTP ratings of problems.

Dom.	#Prob.	#Otter Proofs					#OSHL-U Proofs				
		Total	H	non-Horn			Total	H	non-Horn		
				total	R=0	R>0			total	R=0	R>0
ALG	9	1	0	1	1	0	1	0	1	1	0
ANA	10	0	0	0	0	0	0	0	0	0	0
BOO	64	8	8	0	0	0	1	1	0	0	0
CAT	52	25	20	5	5	0	13	9	4	4	0
COL	150	59	59	0	0	0	1	1	0	0	0
COM	9	6	3	3	3	0	3	2	1	1	0
FLD	143	28	0	28	17	11	68	0	68	21	47
GEO	187	99	2	97	49	48	67	1	66	26	40
GRA	1	1	0	1	1	0	1	0	1	1	0
GRP	503	112	72	40	36	4	48	25	23	22	1
HEN	64	43	43	0	0	0	11	11	0	0	0
HWC	4	1	1	0	0	0	0	0	0	0	0
HWV	73	48	0	48	33	15	26	0	26	20	6
KRS	9	9	1	8	8	0	6	1	5	5	0
LAT	48	6	6	0	0	0	1	1	0	0	0
LCL	422	131	127	4	4	0	13	9	4	4	0
LDA	23	0	0	0	0	0	0	0	0	0	0
MGT	134	86	26	60	42	18	38	2	36	25	11
MSC	11	9	1	8	6	2	4	1	3	1	2
NLP	43	33	8	25	4	21	4	0	4	4	0
NUM	63	18	7	11	9	2	13	2	11	10	1
PLA	30	5	4	1	1	0	0	0	0	0	0
PUZ	58	46	14	32	29	3	40	8	32	28	4
RNG	82	22	19	3	0	3	5	3	2	0	2
ROB	23	1	1	0	0	0	0	0	0	0	0
SET	604	168	2	166	126	40	211	2	209	116	93
SWC	713	145	0	145	27	118	66	0	66	22	44
SWV	13	11	2	9	8	1	11	2	9	8	1
SYN	866	570	338	232	205	27	373	230	143	130	13
TOP	6	6	0	6	5	1	2	0	2	2	0
Total	4417	1697	764	933	636	297	1027	311	716	451	265

Table 5.2: Number of proofs found by Otter and OSHL-U compared over Horn and non-Horn problems. H denotes Horn. R denotes rating.

	Problem Category			
	Total	Horn	non-Horn	non-Horn, Rating>0
#proofs without refinement	900	281	619	218
#proofs with refinement	1027	311	716	265
% increase	14.11	10.68	15.67	21.56

Table 5.3: Increase in number of proofs on different categories of problems due to refinements in OSHL-U implementation.

Chapter 6

Space Efficiency of OSHL-U Strategy and Resolution

6.1 Search Space Efficiency

Execution time is undoubtedly a practical measure of performance for automated theorem proving systems. However, it is not entirely a fair or accurate way of comparing the underlying strategies that different theorem proving systems are based on. This is because execution time depends not only on the strategy but also on how well it is implemented in the theorem proving system; execution time is affected by factors such as the programming language the prover is implemented in, use of efficient data structures, and optimizations performed in implementing the strategy. However, the size of the search space explored by a theorem prover before it finds a proof is not affected by such implementation issues. So search space is a more accurate measure of how efficient a theorem proving strategy is and provides a way of comparing the strategies of different theorem proving systems, independent of their implementation.

6.2 Importance of Storage Space for Theorem Prover Efficiency

For some very hard problems, storage space for inferences can be an important consideration, too. A prover that runs out of space will fail, no matter how fast it is. But a space-efficient method can run for a very long time without exhausting memory, so it could find a proof by running longer.

6.3 Space Efficiency Comparison of OSHL-U and Otter

OSHL-U and Otter strategies are compared here using search spaces and storage spaces. In order to limit the comparison to one between the strategies of OSHL-U and Otter, and

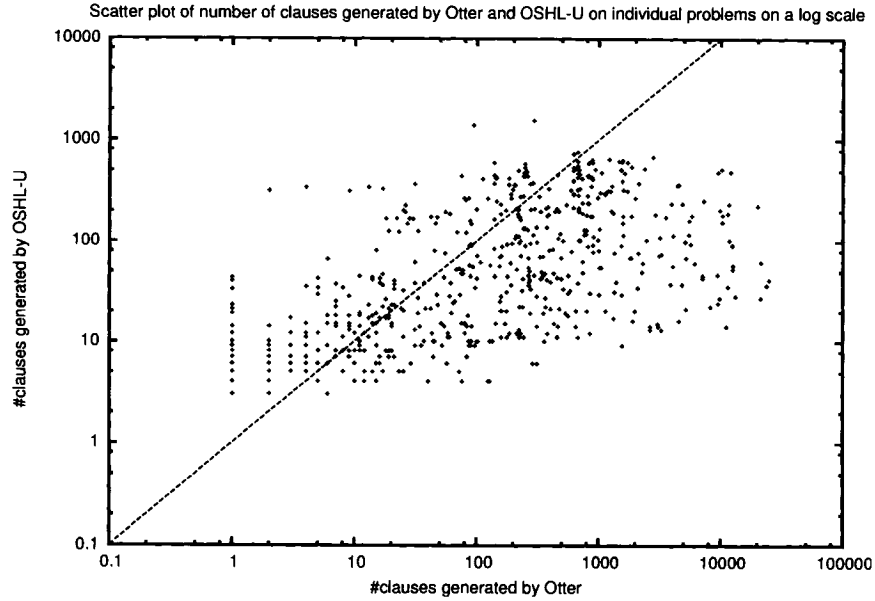


Figure 6.1: Scatter plot showing number of clauses generated by OSHL-U against that of Otter on 827 problems. The plot is on a logarithmic scale. The diagonal line ($x=y$) is shown for comparison. OSHL-U is better on 497 problems, Otter is better on 319 problems, and the provers are the same on 11 problems.

not their implementation, we compared the number of clauses generated by each prover on the same problem, that is, how much of the search space is explored by a prover strategy before it finds a proof. Also, we look at how the storage requirements for Otter and OSHL-U compare using the number of clauses stored by the provers.

We compared on all problems for which we had data for both provers, i.e., problems for which both Otter and OSHL-U succeeded in obtaining proofs within 30 seconds. The OSHL-U statistics used here are from runs with all-negative trivial semantics for all but 32 problems on which proofs were obtained with all-positive trivial semantics. The total number of problems tested on was 4417, which is all the problems in TPTP v2.5.0 that do not have a known status of “satisfiable”. These results are from the runs on the optimized OSHL-U implementation described in Chapter 5. The resolution prover Otter v3.3 [McC94] was also run on the same problems.

We further looked at whether each of these problems is Horn or non-Horn and compared the search spaces and storage spaces on these two categories. We also classified the problems by their TPTP difficulty ratings — rating zero or rating greater than zero — and compared the search spaces on these categories.

Ratio of Clauses Generated by Otter to OSHL-U	Frequency of Occurrence
[0,0.01)	48
[0.01,0.1)	18
[0.1,1)	253
1	11
(1,10)	350
[10,100)	117
[100,1000)	30

Table 6.1: Distribution of the ratio of the number of clauses generated by Otter to that generated by OSHL-U on all individual problems that are proved by both theorem provers. A ratio > 1 means Otter generated more clauses; a ratio < 1 means OSHL-U generated more clauses; a ratio $= 1$ means Otter and OSHL-U generated the same number of clauses.

	All	Horn	non-Horn	R=0	R>0	non-Horn,R>0
Clauses on Otter	708	90	618	357	351	348
Clauses on OSHL-U	104	39	65	78	26	26
Ratio(Otter/OSHL-U)	6.8	2.3	9.5	4.6	13.5	13.5

Table 6.2: Total number of clauses, in thousands, generated by Otter and OSHL-U over problems for which both provers find a proof and the ratio of the number of clauses generated by Otter to that generated by OSHL-U. R denotes TPTP rating.

The number of clauses generated by OSHL-U against that generated by Otter for every problem on which both provers obtained a proof are shown in the scatter plot in Fig. 6.1. The plot is shown on a logarithmic scale. The frequency distribution of the ratios of the number of clauses generated by Otter to that generated by OSHL-U is shown in Table. 6.1. A ratio of 1 means the two provers have the same search space, a ratio less than 1 means Otter has a smaller search space than OSHL-U, and a ratio greater than 1 means OSHL-U has a smaller search space than Otter. Otter has a smaller search space than OSHL-U on 319 problems, OSHL-U has a smaller search space than Otter on 497 problems, and the two provers have the same search space on 11 problems. On the 319 problems on which Otter has a smaller search space than OSHL-U, the average number of clauses generated by Otter is 39.48 and that for OSHL-U is 98.96. On the 497 problems where OSHL-U has a smaller search space, the average number of clauses generated by OSHL-U is 145.36 and that for Otter is 1398.84. The best ratio in favor of Otter is 0 on 44 of the 48 problems shown in the [0, 0.01) range of Table. 6.1; the largest number of clauses that OSHL-U generates on any of these 44 problems is 191. The best ratio in favor of OSHL-U is 794.3. The number of clauses generated on the problem for which the difference between Otter and OSHL-U is highest, that is, Otter generated fewer clauses by the largest margin is 94 on Otter and 1377 on OSHL-U. The number of clauses generated on the problem for which OSHL-U generated fewer clauses than Otter by the largest margin is 24829 on Otter and 41 on OSHL-U. These statistics shows

Domain	All	Horn	nH	R=0	R>0	nH, R>0
ALG	27.2	-	27.2	27.2	-	-
CAT	3.4	2.9	6.9	1.5	7.8	-
COL	2.0	2.0	-	2.0	-	-
COM	0.9	0.6	1.5	0.9	-	-
FLD	117.4	-	117.4	117.2	117.5	117.5
GEO	44.2	0.3	13.3	53.3	53.3	53.3
GRA	0.9	-	0.9	0.9	-	-
GRP	15.3	5.1	17.7	14.1	28.9	28.9
HEN	19.7	19.7	-	19.7	-	-
HWV	3.1	-	3.1	4.1	1.0	1.0
KRS	7.2	0.5	7.5	7.2	-	-
LAT	0.0	0.0	-	0.0	-	-
LCL	2.1	2.9	0.3	2.1	-	-
MGT	2.3	0.4	2.3	1.9	3.7	3.7
MSC	10.8	-	10.8	2.9	20.7	20.7
NLP	1.9	-	1.9	1.9	-	-
NUM	5.9	1.6	6.1	5.9	-	-
PUZ	5.3	6.4	5.2	5.2	5.7	2.2
RNG	220.9	617.8	42.3	617.8	42.3	42.3
SET	6.2	1.3	6.2	4.9	14.0	10.7
SWV	4.4	0.1	4.5	4.2	7.7	7.7
SYN	2.3	1.8	6.8	2.2	29.5	29.5
TOP	1.6	-	1.6	1.6	-	-

Table 6.3: Ratio of total search space of Otter to that of OSHL-U on problems for which both provers find a proof. R denotes TPTP rating. nH denotes non-Horn.

that OSHL-U has smaller search spaces on many problems; and that when Otter generates less clauses than OSHL-U, it does so by not a very large factor, except when Otter generates 0 clauses. It should be noted that in reporting the number of clauses generated, Otter counts 'new' clauses generated only and does not count clauses from the input set of clauses; but OSHL-U includes such clauses, too, in its count. Also, Otter uses hyper-resolution which performs several resolutions in one step without generating the intermediate resolvents. This explains how Otter obtains some proofs by generating 0 clauses.

The sum of the number of clauses generated for all the problems that both provers can prove in 30 seconds, over all domains and the ratio of the sums are shown in Table 6.2. The ratio of the numbers of clauses generated for Otter to those for OSHL-U on individual domains is shown in Table 6.3. The ratios were determined not only for all problems, but also for only those of the problems that are Horn, those that are non-Horn, those that have a rating equal to 0, those that have a rating greater than 0, and those that are non-Horn as well as have a rating greater than 0. A ratio could not be computed when there were

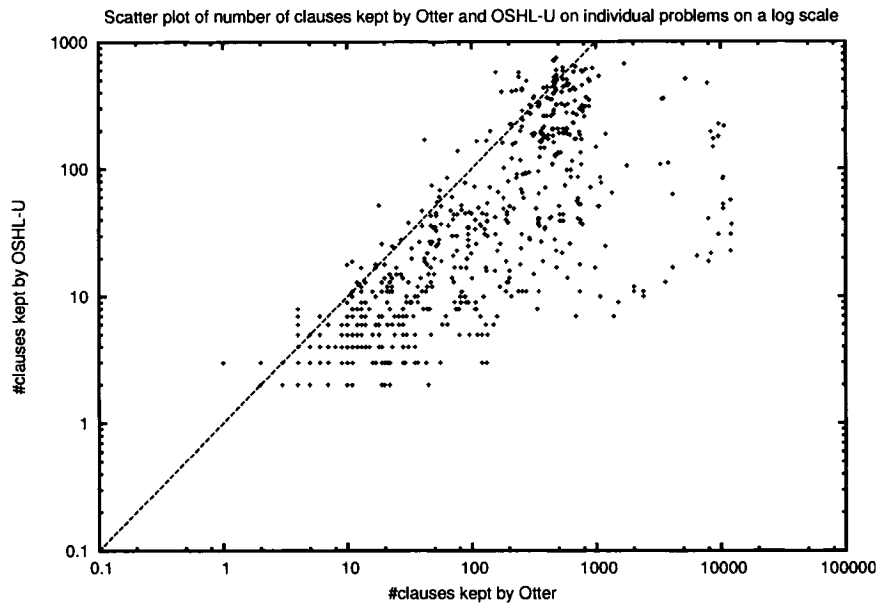


Figure 6.2: Scatter plot showing number of clauses stored by OSHL-U against that of Otter on individual problems. The plot is on a logarithmic scale. The diagonal line ($x=y$) is shown for comparison. OSHL-U is better on 727 problems, Otter is better on 69 problems, and the provers are the same on 31 problems.

no problems in a particular category; such cases are marked as “-” in the table. OSHL-U generates significantly fewer clauses than Otter does in many domains. It is possible that Otter generates significantly fewer clauses than OSHL-U on some problems, but OSHL-U has not run long enough for us to conclude this. The ratios indicate that OSHL-U performs best in comparison to Otter for problems which are non-Horn and which have ratings greater than 0. On some domains, the number of proofs that both provers get are so few that such trends are not observable. OSHL-U even seems to generate smaller search spaces than Otter for Horn clauses on several domains.

In order to compare the space requirements of each prover, we compare the number of clauses retained by each prover on each problem. For OSHL-U, this is the maximum of the number of clauses in the ascending sequence at any point during execution. These comparisons were done in a way similar to the comparison of search spaces, by using the number of clauses stored by each prover. The number of clauses stored by OSHL-U is plotted against the number of clauses stored by Otter in the scatter plot in Fig. 6.2 and the frequency of occurrence of their ratios are shown in Table 6.4. These show that on most of the problems compared, Otter uses significantly more storage space than OSHL-U and on a few problems, it uses less space than OSHL-U. The comparison of storage space used over all domains and on individual domains is shown in Table 6.5 and Table 6.6. These data also indicate that OSHL-U stores

Ratio of Clauses Stored by Otter to OSHL-U	Frequency of Occurrence
[0,0.01)	0
[0,0.1)	0
[0.1,1)	69
1	31
(1,10)	605
[10,100)	100
[100,1000)	22

Table 6.4: Distribution of the ratio of number of clauses stored by Otter to that stored by OSHL-U on all individual problems that are proved by both provers. A ratio > 1 means Otter stored more clauses; a ratio < 1 means OSHL-U stored more clauses; a ratio $= 1$ means Otter and OSHL-U stored the same number of clauses.

	All	Horn	non-Horn	R=0	R>0	non-Horn,R>0
Clauses on Otter	423	81	342	230	193	192
Clauses on OSHL-U	92	37	55	67	25	25
Ratio(Otter/OSHL-U)	4.6	2.2	6.2	3.4	7.7	7.7

Table 6.5: Total number of clauses, in thousands, stored by Otter and OSHL-U over problems for which both provers find a proof and the ratio of the number of clauses stored by Otter to that stored by OSHL-U. R denotes TPTP rating.

significantly fewer clauses than does Otter.

It should be noted that OSHL-U generates far fewer clauses per second than Otter, which has a more sophisticated implementation. The tests were executed on a Pentium 4 1.7 GHz processor with 256 MB of memory running Linux; on this computer, Otter can generate clauses numbering in the order of 10000 per second, while OSHL-U can generate clauses numbering in the order of 100 per second. OSHL-U is implemented in OCaml and Otter in C. OSHL-U does not have term rewriting or any special support for handling equality; neither does it have any special data structure support for speeding up the implementation. OSHL-U was not designed for Horn problems and is not necessarily better than UR resolution on this kind of problem. The results suggest that the performance of OSHL-U relative to resolution improves as the problems become harder and non-Horn.

6.4 Conclusion

We compare the strategy of OSHL-U to Otter, independent of how efficiently the provers are implemented. We consider the number of clauses generated as a good metric for comparison. However, it should be noted that Otter uses term-rewriting which reduces the search space on some problems. Otter generates in all over 6 times as many clauses as OSHL-U

Domain	All	Horn	non-Horn	R=0	R>0	non-Horn, R>0
ALG	6.7	-	6.7	6.7	-	-
CAT	2.4	2.0	4.0	1.9	3.7	-
COL	3.5	3.5	-	3.5	-	-
COM	0.9	0.8	1.1	0.9	-	-
FLD	88.2	-	88.2	104.9	81.5	81.5
GEO	8.9	2.0	9.0	9.9	8.7	8.7
GRA	3.4	-	3.4	3.4	-	-
GRP	4.1	4.7	3.9	3.7	6.4	6.4
HEN	4.2	4.2	-	4.2	-	-
HWV	2.8	-	2.8	3.7	0.8	0.8
KRS	3.2	1.3	3.4	3.2	-	-
LAT	5.7	5.7	-	5.7	-	-
LCL	4.4	5.5	1.6	4.4	-	-
MGT	2.4	1.6	2.4	2.5	2.0	2.0
MSC	10.1	-	10.1	1.2	20.5	20.5
NLP	1.9	-	1.9	1.9	-	-
NUM	6.9	2.7	7.1	6.9	-	-
PUZ	4.5	6.0	4.3	4.4	5.2	1.7
RNG	46.1	129.6	12.7	129.6	12.7	12.7
SET	4.4	2.9	4.4	3.3	10.7	10.7
SWV	2.5	1.0	2.6	2.4	3.0	3.0
SYN	2.2	2.1	4.4	2.1	8.6	29.5
TOP	5.0	-	5.0	5.0	-	-

Table 6.6: Ratio of total storage space of Otter to that of OSHL-U on problems for which both provers find a proof. R denotes TPTP rating.

on the problems for which both found proofs, and on non-Horn problems, Otter generates over 10 times as many clauses. On problems of rating greater than zero, Otter generates over 13 times as many clauses overall. Otter stores about 4 times as many clauses on all problems, twice as many clauses on Horn problems, and 6 times as many clauses on non-Horn problems. It appears that the search space advantage of OSHL-U over Otter is greater on non-Horn problems and on problems of rating greater than zero, which are presumably harder for resolution. While it would be good to be able to also find proofs for those problems that are already easy for resolution, our real interest is in finding more efficient strategies for problems that are not. One of our objectives is to investigate the instance-based approach on problems that are difficult for resolution provers, in an effort towards making automated theorem provers, in general, stronger. It may be that OSHL-U has better asymptotic behavior than resolution – this would imply that for problems that are harder for resolution, the performance of OSHL-U would get better relative to Otter.

.

Chapter 7

Comparison of Theorem Provers with Different Inference Rates

In this chapter, some techniques for comparing theorem provers with different rates of inference generation are presented. These techniques make it possible to carry out such comparisons independent of the inference rate and of the speed of the machines that the provers execute on. Using these methods, we compare Otter and OSHL-U, which differ in inference generation rates by several orders of magnitude. The results indicate that OSHL-U is superior to Otter on certain kinds of problems, in spite of having a lower inference rate than Otter.

7.1 Motivation

Often in automated reasoning work, comparisons have to be made between theorem provers that differ greatly in their prover strategy, implementation and rates of generating inferences. For example, one theorem prover may be implemented in a low-level language with efficient data structures while another theorem prover may be implemented in a higher level language and with less efficient data structures. The former prover would generally have a much higher inference generation rate. It is also possible that one theorem prover uses inference strategies that are harder to implement than those of another prover. So even with equivalent implementations, the rate of inference generation of the two provers could differ considerably. All this makes it difficult to get an idea of the relative merits of the two provers in a way that factors out the differences in their implementation. But it is important to have an estimate of the relative powers of the two theorem provers so that one can decide whether it would be worthwhile to spend additional effort optimizing the less well implemented prover. There is, therefore, the need to have techniques that can compare theorem proving capabilities independent of inference rate.

7.2 Comparing Theorem Provers with Different Inference Rates

The functions defined in this section are useful in estimating theorem prover efficiency in terms of time and space, while factoring out the differences in inference rate and efficiency of the implementation.

Informally speaking, the time efficiency function measures the number of proofs obtained with the number of inferences generated; the space efficiency function measures the total amount of space used by a prover as a function of the number of proofs obtained. If a prover were run in parallel on all the problems, then the number of proofs obtained with increasing number of inferences is given by the time efficiency function. Similarly, if a prover were run on all the problems in parallel, then the maximum total space requirement of the prover with increasing number of proofs obtained is given by the space efficiency function. In computing these functions, the assumption is that inferences are generated by a prover at the same rate on all problems. This assumption may not be justified for problems on which the prover has to do more computation in order to generate inferences.

7.2.1 Time Efficiency

Time efficiency for a prover P on a given set of problems can be defined by a function $f_P: N \rightarrow N$, such that $f_P(n)$ is the number of proofs obtained after P is allowed to generate n inferences on each problem in the set. This gives some idea of the asymptotic behavior of the prover as n becomes large. f_P can be determined experimentally for all values in the range $[0, n]$ by running the prover for n inferences on each problem and then computing f_P for values less than n . Ideally, this data would be collected by running a theorem prover for some maximum number of inferences on every problem. However, mechanisms to run for a bounded number of inferences are not provided in automated theorem proving systems, in general. Rather some mechanism that allows a prover to be run for a bounded amount of execution time is common. Assuming that inferences increase monotonically with time, the data for computing the time-efficiency function can be collected by running for some large execution time.

If two provers P_1 and P_2 were implemented equally well, then presumably their inference rates would be identical, so the functions f_{P_1} and f_{P_2} would give a measure of the relative numbers of proofs obtained by the two provers in the same execution time. It is possible that one prover uses a method that is harder to implement, so that even with equivalent implementations, the inference rate of prover P_2 is r times slower than that of prover P_1 . Then instead of comparing the functions $f_{P_1}(n)$ and $f_{P_2}(n)$, one could compare $f_{P_1}(rn)$ and $f_{P_2}(n)$ to get an idea of their relative behavior. This could be done for more than one value of r to see how close the inference rate of P_2 would have to be to that of P_1 in order for the

provers to have comparable behavior.

7.2.2 Space Efficiency

A similar method can be used to compare storage space requirements for theorem provers. Let us define space efficiency for a prover P on a given set of problems by a function $g_P: N \rightarrow N$ as follows. Let t be the time taken by prover P to obtain n proofs. Then $g_P(n)$ denotes the sum of space required to generate each proof, that is obtained within time t by P . As in the case of the time efficiency functions, one can compute $g_{P_1}(n)$ and $g_{P_2}(n)$ for various values of n and estimate the relative behavior of two provers P_1 and P_2 in terms of their space requirements as n becomes large. The functions g_{P_1} and g_{P_2} give us the space the provers require as a function of the number of proofs generated. Thus, these serve as a measure of how efficient the provers are at using space. Such a measure is also independent of the inference rates of the provers.

The definitions of the f and g functions are not analogous. The space efficiency function g adds space, but the time efficiency function f does not add times. This is because when problems are run in parallel, space is additive but time is not.

7.3 Relative Efficiency of OSHL-U and Otter

OSHL-U is implemented in Ocaml and Otter in C with better data structures. These differences are reflected in the provers' different rates of inference. While Otter is capable of generating clauses on the order of 10,000 per second, OSHL-U can generate clauses on the order of 100 per second. OSHL-U could be improved by adding term rewriting, semantics, special methods for handling equality, and better data structures. But to know whether such effort would be worthwhile, there is a desire to compare the provers in a way that factors out the differences in their inference rates.

Recall that we compared execution times to obtain proofs and the number of proofs obtained within fixed CPU time limits; these are highly dependent on inference rate. We also compared strategy using the search space and storage space requirements. However, even this comparison is dependent on inference rate because Otter can generate more clauses than OSHL-U in 30 seconds.

Based on the data from the tests with the theorem provers described in previous Chapters, we compute the time efficiency and space efficiency functions for Otter and OSHL-U. These give us an interesting way of looking at the two provers which have very different inference rates. However, because the runs were limited by execution time and not by the number

of inferences generated, the functions computed here are an approximation to the actual functions. On the assumption that number of inferences generated increase monotonically with execution time, these approximations still give us some estimate of the relative powers of Otter and OSHL-U.

7.4 Time Efficiency

7.4.1 Comparison over all, Horn and non-Horn Problems

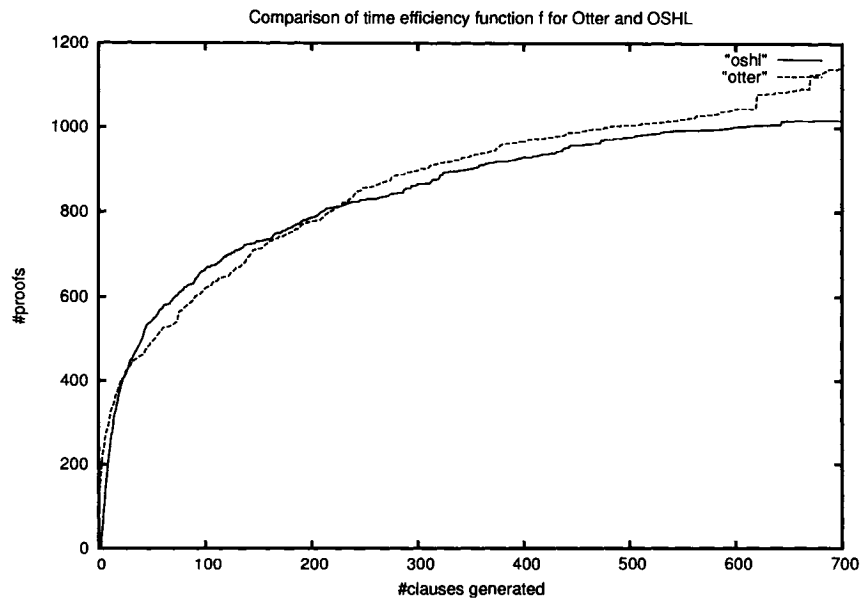


Figure 7.1: Comparison of time efficiency functions for Otter and OSHL-U on all problems. X-axis represents the number of clauses generated and Y-axis represents the number of proofs obtained.

The time efficiency functions for Otter and OSHL-U were computed on all the problems, on only the Horn problems, and on only the non-Horn problems. Figure 7.1 shows the time efficiency function computed for OSHL-U and Otter over all problems. This indicates that Otter has slightly better performance than OSHL-U as evidenced by the higher curve for f_{Otter} . Also, f_{Otter} shows a more steeply rising curve while f_{OSHL-U} flattens out indicating a saturation in the number of proofs.

Figure 7.2 and Figure 7.3 show the time efficiency functions for OSHL-U and Otter on Horn problems and on non-Horn problems, respectively. On Horn problems, Otter is clearly better than OSHL-U as seen by a significantly higher curve for f_{Otter} than for f_{OSHL-U} .

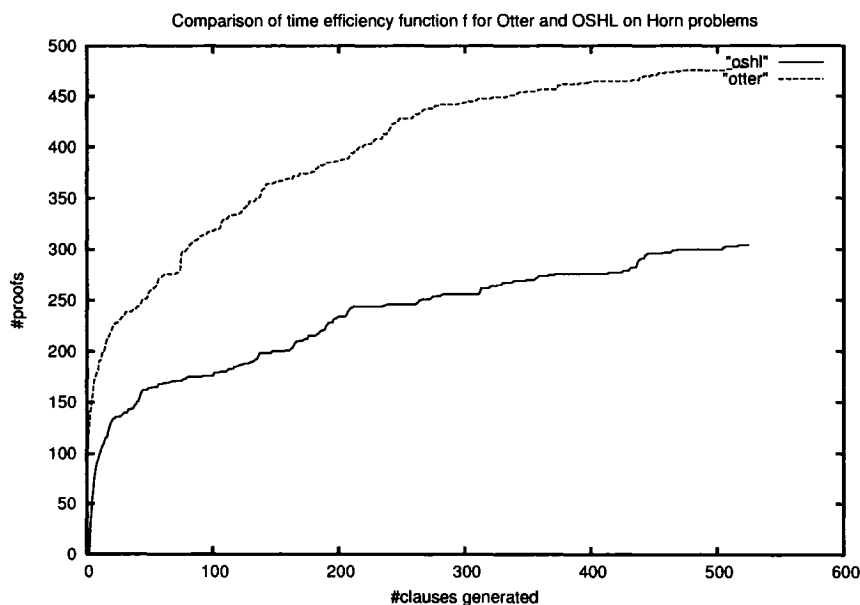


Figure 7.2: Comparison of time efficiency functions for Otter and OSHL-U on Horn problems. X-axis represents the number of clauses generated and Y-axis represents the number of proofs obtained.

Otter performs better than OSHL-U on Horn problems more significantly than on all the problems. On non-Horn problems, however, f_{OSHL-U} has a significantly higher curve than f_{Otter} indicating that OSHL-U performs much better than Otter.

Figure 7.1 showed f_{Otter} and f_{OSHL-U} to be comparable over the range plotted. However, f_{Otter} seems to be rising while f_{OSHL-U} flattens out. If f_{Otter} continues to rise in the same way with increasing numbers of clauses generated, then this indicates that performance of Otter gets asymptotically better than that of OSHL-U with greater number of inferences generated. So we looked at f_{Otter} further. Because of the higher inference rate, Otter generated many more inferences than OSHL-U within the 30 second CPU time limit; we used the data to plot f_{Otter} over a larger range of inferences as shown in Figure 7.4. As this figure shows, the number of proofs obtained by Otter seems to reach saturation as seen in the flattening out of f_{Otter} . The runs are limited by CPU time and inference generation rates are not strictly uniform on all problems; even with the same prover, inference generation rate is often problem-class specific. Therefore, at the higher values in the range over which we plot the function, fewer problems may have run for as many inferences so our approximation could deviate from the ideal function by a larger amount. This could account for the levelling off in the number of proofs, as observed in the case of Otter as well as of OSHL-U.

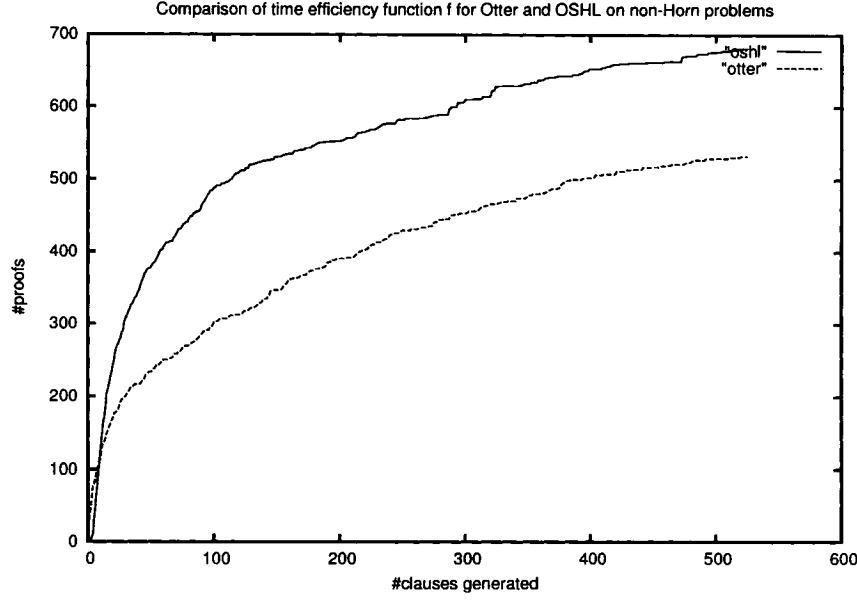


Figure 7.3: Comparison of time efficiency functions for Otter and OSHL-U on non-Horn problems. X-axis represents the number of clauses generated and Y-axis represents the number of proofs obtained.

7.4.2 Performance on Field Theory and Group Theory Categories

To further investigate the difference in behavior of the provers on Horn and non-Horn problems, we allowed OSHL-U to run for 300 seconds on each of the problems in the TPTP domains FLD(field theory) and GRP(group theory); this gives us a longer range over which to compare f_{Otter} and f_{OSHL-U} . These domains are of interest because all the problems in FLD are non-Horn problems and many of the problems in GRP are Horn. Figure 7.5 shows f_{Otter} and f_{OSHL-U} on FLD problems. Figure 7.6 shows f_{Otter} and f_{OSHL-U} on GRP problems. These figures show OSHL-U to be superior on FLD problems and Otter to be superior on GRP problems. These observations concur with our general observations on performance of Otter and OSHL-U over all Horn and all non-Horn problems, in general.

7.5 Space Efficiency

We compare space efficiency of Otter and OSHL-U in a similar way as we did time efficiency. Figure 7.7 shows the comparison of the space efficiency functions g_{OSHL-U} and g_{Otter} on all problems. The overall space efficiency of the two provers is similar.

Figure 7.8 shows the comparison of space efficiencies of Otter and OSHL-U on Horn problems. Otter uses much less space than OSHL-U on these problems. g_{Otter} and g_{OSHL-U}

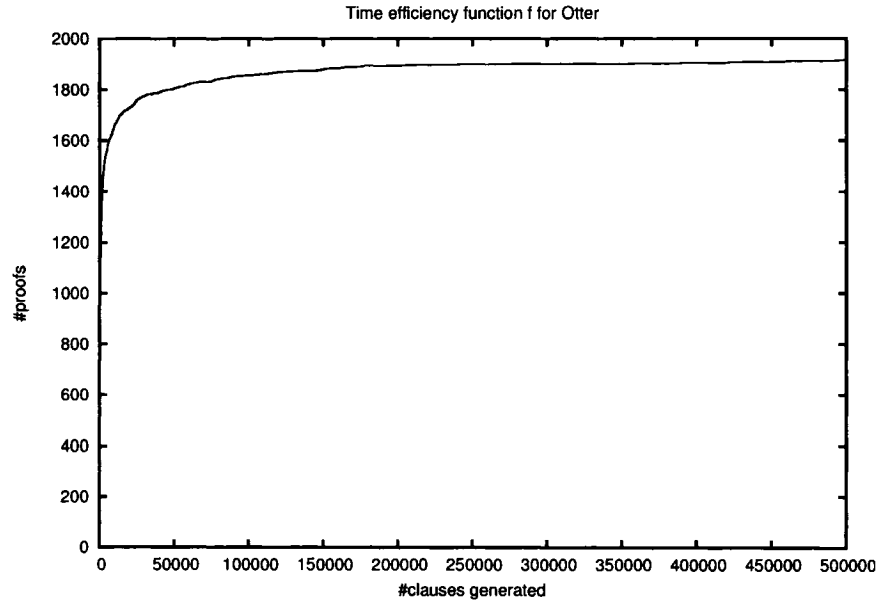


Figure 7.4: Time efficiency function for Otter over a long range. X-axis represents the number of clauses generated and Y-axis represents the number of proofs obtained.

have similar space efficiencies over smaller number of proofs; in fact, OSHL-U seems even more space-efficient than Otter. However, space requirements of OSHL-U rise sharply with increasing number of proofs, significantly exceeding that of Otter. Sometimes, a prover uses extra clauses on “starting up”; this would show up as a sharp initial rise in the plot of the g function, but a less sharp rise subsequently. In this case, we see a similar effect with g_{Otter} . the g_{Otter} and $g_{\text{OSHL-U}}$ curves diverge rapidly with increasing number of proofs, meaning that for generating the same number of proofs on Horn problems, the space requirements of OSHL-U grow more and more rapidly compared to that of Otter as more and more proofs are generated.

Figure 7.9 shows the comparison of space efficiencies of Otter and OSHL-U on non-Horn problems. On these problems, Otter uses more and more space than OSHL-U with increasing number of proofs. So OSHL-U is significantly more space-efficient than Otter on non-Horn problems.

7.6 Discussion

The results obtained with the different techniques of comparing prover efficiency in this and the previous Chapters may seem somewhat contradictory. The fact that Otter obtains more proofs than OSHL-U within a fixed time limit, of course, favors Otter as a more time-

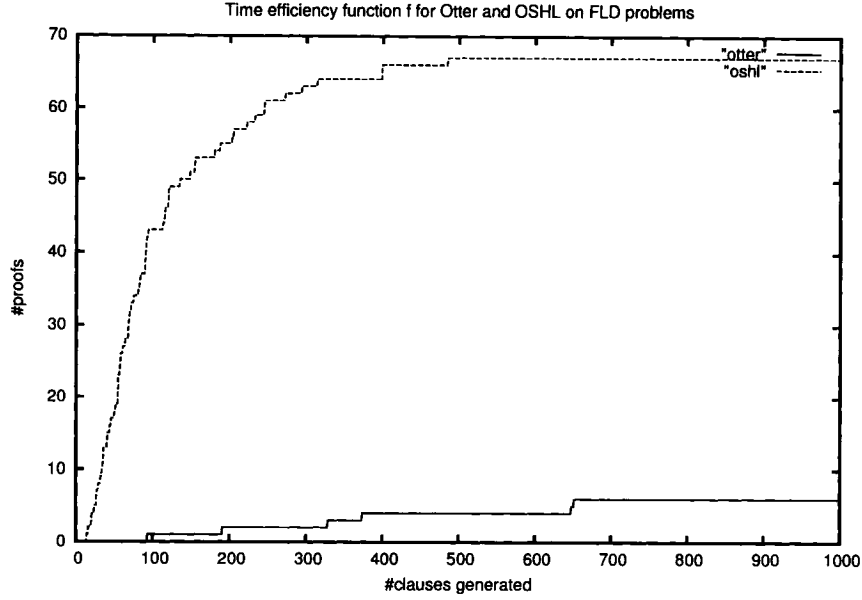


Figure 7.5: Comparison of time efficiency functions for Otter and OSHL-U on FLD problems. X-axis represents the number of clauses generated and Y-axis represents the number of proofs obtained.

efficient prover. But this is hardly surprising considering that Otter generates inferences at a much higher rate than does OSHL-U. However, the fact that OSHL-U is able to get more than half the proofs even with a much lower inference rate tends to favor OSHL-U. Figure 7.1 suggests that Otter may rapidly outperform OSHL-U as inferences increase, but the flattening out of f_{Otter} in Figure 7.4 puts this in question. Also, OSHL-U obtains more proofs than Otter on set theory (SET) and field theory (FLD) problems within the 30 second time limited runs suggesting OSHL-U is already more time-efficient than Otter on non-Horn problems without equality despite a lower inference rate than Otter. The relative number of proofs in the Horn and non-Horn categories also supports that OSHL-U performs better in relation to Otter on non-Horn problems. The comparison based on the time efficiency functions clearly favors OSHL-U as being more time-efficient on non-Horn problems. The number of problems the time efficiency functions were compared over is not very large, so this result can not be conclusively generalized over all non-Horn problems. Based on the time efficiency functions, Otter is favored as somewhat more time-efficient overall and especially on Horn problems. These conclusions are further supported by looking at the time efficiency functions on two of the TPTP domains — group theory and field theory. On field theory (FLD) problems, all of which are non-Horn, OSHL-U appears to be more time-efficient, while on group theory (GRP) problems, many of which are Horn, Otter appears to be more time-efficient.

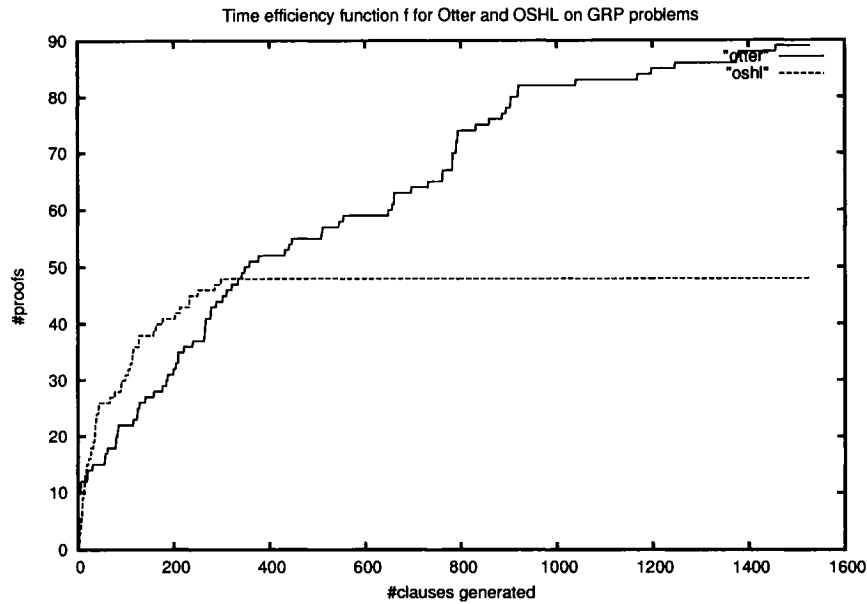


Figure 7.6: Comparison of time efficiency functions for Otter and OSHL-U on GRP problems. X-axis represents the number of clauses generated and Y-axis represents the number of proofs obtained.

The count of the number of clauses stored favors OSHL-U as a more space-efficient prover. The comparison based on the space efficiency functions also favors OSHL-U as being more space-efficient overall and especially so on non-Horn problems.

These results show that it is not always easy to say which of two automated reasoning methods is better overall. This is especially so when the implementations differ greatly in inference rate and level of optimization. Using a single method of comparison could be misleading, so several methods of comparison were used to get a more complete picture. The time efficiency and space efficiency functions in this chapter provide some additional techniques for carrying out such comparisons.

On some problems OSHL-U, uses much less space than Otter and many fewer inferences. And, on some problems, OSHL-U even uses less time than Otter. These results are independent of inference rate because a faster inference rate for OSHL-U will not affect space but will only increase the time advantage of OSHL-U on some problems. Likewise, the fact that a propositional prover not using true unification can prove over half of the problems that Otter proves is a fact that is interesting in itself and is not invalidated by OSHL-U's slow inference rate.

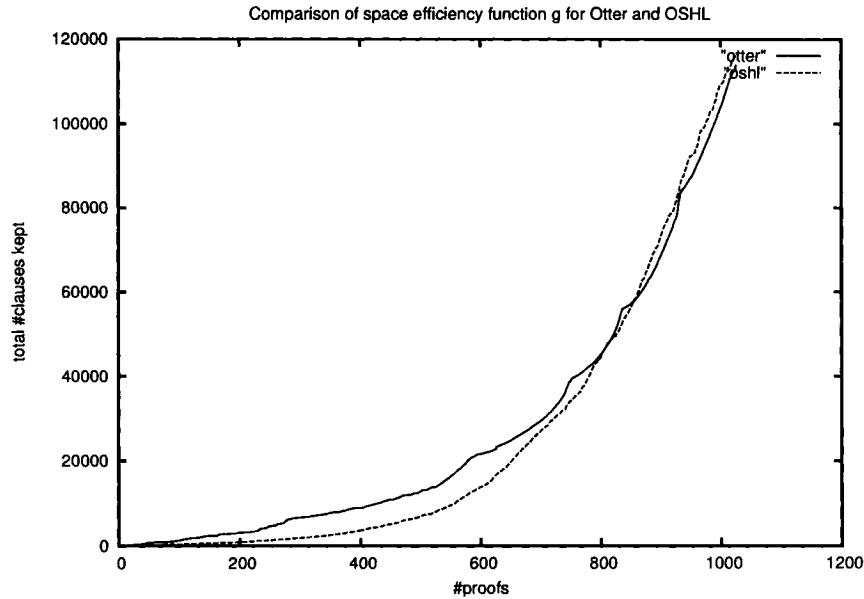


Figure 7.7: Comparison of space efficiency of Otter and OSHL-U over all problems. X-axis represents the number of proofs obtained and Y-axis represents the total number of clauses kept.

The fact that OSHL-U uses much less space on some problems, and gets more proofs than Otter in 30 seconds on FLD and SET, suggests that OSHL-U is already superior to resolution on certain problems. This means that OSHL-U may already be a useful tool for research and applications because it may be able to solve a class of problems that resolution cannot. In support of this, OSHL-U proved at least one problem of TPTP rating 1.00, which means that as of the date of the TPTP release used, no other prover could prove it. Of course, further enhancements to OSHL-U such as term rewriting and semantics should only increase its utility.

Even a prover that is slower overall and gets fewer proofs can still be superior on certain kinds of problems and thus be a useful tool for research and applications. The results obtained with OSHL-U and Otter indicate that propositional approaches to proving first-order problems are worthwhile, at least for certain kinds of problems. It is possible that a combination of propositional and resolution techniques could be superior to using resolution alone.

7.7 Drawbacks of the Comparison Techniques

Otter has a simple control structure that makes it possible for us to consistently apply a uniform strategy. However, there are some drawbacks to applying the f and g functions

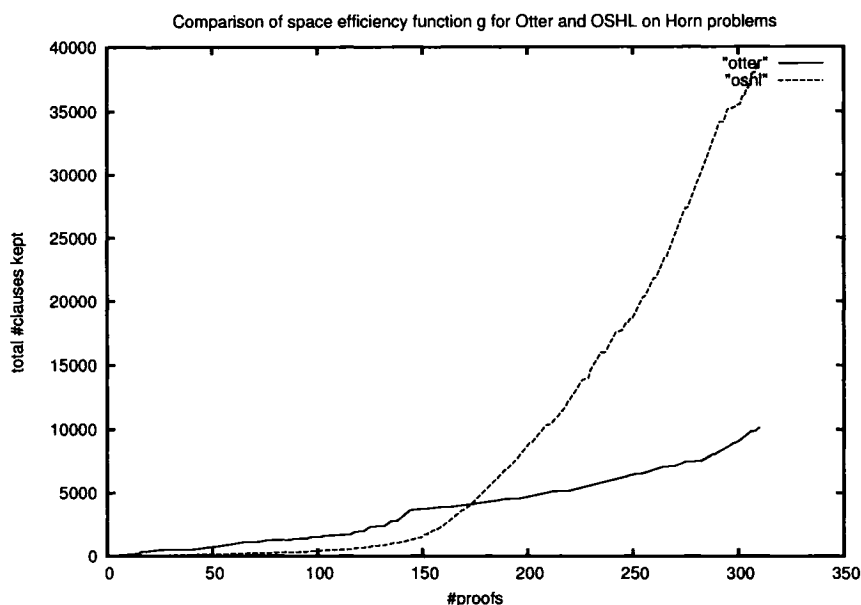


Figure 7.8: Comparison of space efficiency of Otter and OSHL-U over Horn problems. X-axis represents the number of proofs obtained and Y-axis represents the total number of clauses kept.

described to modern theorem provers. Modern provers are often run taking a fixed time and/or space limit into consideration. For example, when there is little time left a modern prover will often throw away large clauses because these are probably not going to contribute to a proof that can be determined within the time limit. When this is the case, it is difficult to determine how many inferences a prover takes to prove a result, because this may depend on external parameters. Similar issues occur in space usage, particularly for provers that try to keep all often-used information in fast memory and for provers that try to exploit all the fast memory that is available to them, only getting rid of retained results when this memory is exhausted.

Modern provers also are tuned to particular problem classes, and may exhibit significantly better performance after tuning. Also, provers use strategy scheduling whereby the available time limit is sliced between several different strategies instead of using a single uniform strategy throughout. These factors make it difficult to have a straightforward evaluation of strategy in modern theorem provers using the efficiency functions.

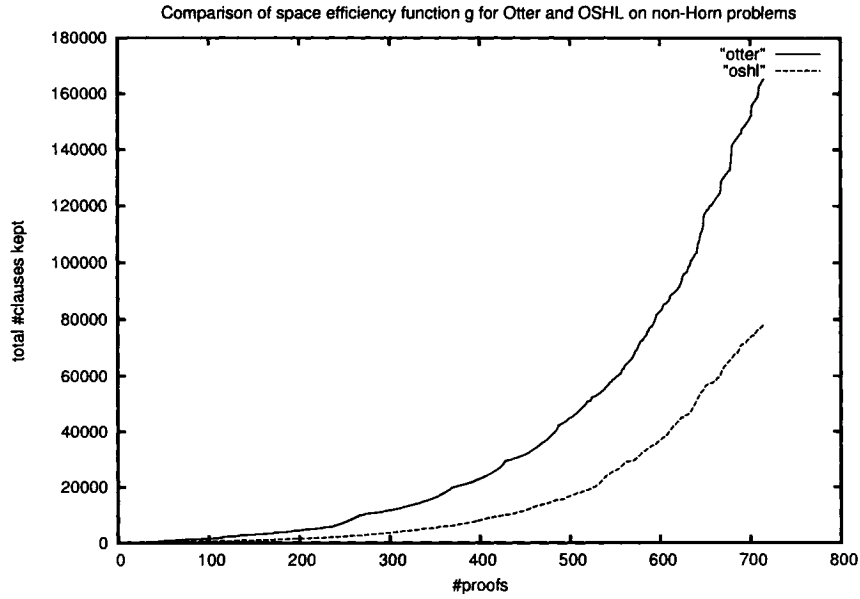


Figure 7.9: Comparison of space efficiency of Otter and OSHL-U over non-Horn problems. X-axis represents the number of proofs obtained and Y-axis represents the total number of clauses kept.

7.8 Conclusion

The time efficiency and space efficiency functions indicate that Otter is more time-efficient and more space-efficient than OSHL-U on Horn problems, while OSHL-U is more time-efficient and more space-efficient than Otter on non-Horn problems. This is observed on Horn and non-Horn problems overall as well as in the individual domains of GRP and FLD, which contain mostly Horn problems and mostly non-Horn problems, respectively.

Chapter 8

Performance of OSHL-U on Problems Requiring Definition Expansion

8.1 Introduction

An important problem in trying to prove a theorem is when to expand definitions [Wos88]. Definitions represent concepts in a theory and are important in many proofs. For example, theorems in set theory can be proven largely by expanding definitions. However, expanding definitions increases the search space a theorem prover has to look at and can degrade the performance of the prover. Moreover, quantifiers are eliminated during translation to the clause normal form, so it is difficult for clause form theorem provers for first-order logic to replace a predicate by its definition when the definition involves new quantifiers. It was shown by Bledsoe [Ble97] that clause form provers do not perform well on problems involving definitions. Because the performance of a theorem prover can be affected by the way in which it handles definitions, it is important to handle definitions well. In this Chapter, we look at how proofs requiring definition expansion are handled by OSHL-U. We compare the performance of OSHL-U on some set theory problems requiring definition expansion to that of resolution, using the resolution theorem provers, Otter [McC90] and Vampire [RV02]. We also compare to E-SETHEO [SW00], a prover based on model elimination, and to DCTP [LS01], which is an implementation of the disconnection calculus described in [Bil96]; DCTP is an example of a prover based on propositional strategies. All these are leading modern automated theorem provers that have outperformed many other provers in the most recent CASC competitions; these provers also represent powerful provers that use several different approaches. The implementation of OSHL-U is a general theorem prover and does not have any special rules for definition detection or replacement. In that, it differs from the earlier approach of [PZ99] to using replacement rules for definitions with OSHL that only worked for definition expansion. One of the unit rules implemented in OSHL-U, case analysis, simulates definition expansion by case analysis of the literals in a clause.

8.2 Survey of Approaches to Definition Expansion

Some of the approaches to handling definitions used in the past have involved special rules for replacing predicates by their definitions. The method used in [PG86] was to replace predicates by their definitions before translating to clause form, and then use a structure-preserving clause form translation and a variant of locking resolution. In [PP91], replacement rules were applied in a way similar to term rewriting rules, where the replacement rules and their orientation were selected by the user. Both these methods were effective with set theory problems. In [LP94], replacement rules were selected by the user, with extra features added for difficult proofs. This was improved upon in CLIN-S [PC94] with automatic selection of replacement rules for definition expansion and contraction. The RRTP theorem prover [PP97] also used automatic selection of replacement rules, to perform different kinds of replacement at the same time. It was highly efficient for problems involving concept description languages [PP98]. An extensive study of replacement rules is given in [Par97], and a variant of those rules was used in a fully automatic way in [PZ99]. In [GOP93], criteria are proposed for eliminating defined concepts. Definition expansion in a first-order combinatory logic setting are considered in [DHK98], and a complete approach to higher-logic logic in this way is given. However, approaches involving higher-order logic can explicitly represent quantifiers in definitions. Quaife [Qua92] obtained many set theory proofs using the general first-order prover Otter with special settings and particular clause weights to guide the search. In [BA98], both the original as well as the expanded clauses are retained, and this approach is incorporated into a higher-order theorem prover. Theorems with both universal and existential quantifiers are handled in [BF91]; our case is simpler with only universally quantified variables.

8.3 Experiments

We describe the results of our experiments on OSHL-U, Otter 3.3, Vampire 7.0, E-SETHEO and DCTP 1.3 with a set of problems in set theory. The sets of theorems used to test the provers are as follows.

p1 left associative. $S1 \cup S2 \cup \dots \cup Sn = Sn \cup S(n-1) \cup \dots \cup S1$ for various n , with both sides associated to the left.

p1 right associative. $S1 \cup S2 \cup \dots \cup Sn = Sn \cup S(n-1) \cup \dots \cup S1$ for various n , with the left side associated to the left and the right side associated to the right.

p2 left associative. $S1 \cup S2 \cup \dots \cup Sn = S1 \cup S2 \cup \dots \cup Sn \cup S1 \cup S2 \cup \dots \cup Sn$ for various n , with both sides associated to the left.

p2 right associative. $S1 \cup S2 \cup \dots \cup Sn = S1 \cup S2 \cup \dots \cup Sn \cup S1 \cup S2 \cup \dots \cup Sn$ for various n , with the left side associated to the left and the right side associated to the right.

p3. $S1 \cup S2 \cup \dots \cup Sn = S1 \cup S2 \cup \dots \cup Sn$ for various n , with the left side associated to the left and the right side associated to the right.

p4 left associative. $S1 \cap S2 \cap \dots \cap Sn = Sn \cap S(n-1) \cap \dots \cap S1$ for various n , with both sides associated to the left.

p4 right associative. $S1 \cap S2 \cap \dots \cap Sn = Sn \cap S(n-1) \cap \dots \cap S1$ for various n , with the left side associated to the left and the right side associated to the right.

p5 left associative. $S1 \cap S2 \cap \dots \cap Sn = S1 \cap S2 \cap \dots \cap Sn \cap S1 \cap S2 \cap \dots \cap Sn$ for various n , with both sides associated to the left.

p5 right associative. $S1 \cap S2 \cap \dots \cap Sn = S1 \cap S2 \cap \dots \cap Sn \cap S1 \cap S2 \cap \dots \cap Sn$ for various n , with the left side associated to the left and the right side associated to the right.

p6. $S1 \cap S2 \cap \dots \cap Sn = S1 \cap S2 \cap \dots \cap Sn$ for various n , with the left side associated to the left and the right side associated to the right.

For the problems, the definitions of \subset (subset), set equality, \cup (set union), and \cap (set intersection) were supplied when needed. These definitions are as follows.

Subset	$\{\text{not}(\text{memb}(E, A)), \text{not}(\text{subset}(A, B)), \text{memb}(E, B)\}.$ $\{\text{subset}(A, B), \text{memb}(\text{memb_of_1_not_of_2}(A, B), A)\}.$ $\{\text{not}(\text{memb}(\text{memb_of_1_not_of_2}(A, B), B)), \text{subset}(A, B)\}.$
Equality	$\{\text{not}(\text{equal}(A, B)), \text{subset}(A, B)\}.$ $\{\text{not}(\text{equal}(B, A)), \text{subset}(A, B)\}.$ $\{\text{not}(\text{subset}(A, B)), \text{not}(\text{subset}(B, A)), \text{equal}(B, A)\}.$
Union	$\{\text{not}(\text{memb}(A, \text{union}(B, C))), \text{memb}(A, B), \text{memb}(A, C)\}.$ $\{\text{not}(\text{memb}(A, B)), \text{memb}(A, \text{union}(B, C))\}.$ $\{\text{not}(\text{memb}(A, B)), \text{memb}(A, \text{union}(C, B))\}.$
Intersect	$\{\text{not}(\text{memb}(C, \text{intersect}(A, B))), \text{memb}(C, A)\}.$ $\{\text{not}(\text{memb}(C, \text{intersect}(A, B))), \text{memb}(C, B)\}.$ $\{\text{not}(\text{memb}(C, A)), \text{not}(\text{memb}(C, B)), \text{memb}(C, \text{intersect}(A, B))\}.$

We ran the problems on OSHL-U, Vampire 7.0, Otter 3.3, E-SETHEO and DCTP 1.3. We performed the Vampire, E-SETHEO and DCTP runs on SystemOnTPTP [SS98a] with a time limit of 300 seconds on each problem for each prover. The command-line arguments used for DCTP were ‘-negpref -complexity -fullrewrite -alternate -resisol’. Vampire and E-SETHEO were run as they are provided by default on SystemOnTPTP without any command-line arguments. Both Vampire and E-SETHEO use the technique of strategy scheduling

[Tam97, SW99] whereby a problem is examined to identify several potential good strategies for solving the problem, some CPU time is assigned to each strategy and then these strategies are attempted one after the other in some order till a solution is found. We performed the Otter and OSHL-U runs on a Pentium 4 1.7 GHz processor with 256 MB of memory running Linux. We executed OSHL-U with all-positive trivial semantics (back chaining) with a timeout of 300 seconds on each problem. We performed the runs on Otter with several different settings as follows, using 600 seconds on each problem for each setting.

- autonomous mode (“auto” flag) with only the negation of the theorem in the set of support
- negative hyper-resolution (“neg_hyper_res” flag) with only the negation of the theorem in the set of support
- positive hyper-resolution (“hyper_res” flag) with all the positive clauses and the negation of the theorem in the set of support
- binary resolution with only the negation of the theorem in the set of support

Tables 8.1 - 8.10 show a comparison of the execution time, clauses generated and clauses kept on OSHL-U, Otter and Vampire and the execution time on E-SETHEO and DCTP on the sets of problems described. Note that the set of problems, p1 left associative ($S1 \cup S2 \cup \dots \cup Sn = Sn \cup S(n-1) \cup \dots \cup S1$ with both sides associated to the left), starts at $n = 2$, while the set of problems, p1 right associative ($S1 \cup S2 \cup \dots \cup Sn = Sn \cup S(n-1) \cup \dots \cup S1$ with the left side associated to the left and the right side associated to the right), starts at $n = 3$. This is because the $n = 2$ case for p1 right associative is the same as for p1 left associative. Similar reasoning applies to the sets, p2 left associative and p2 right associative, p4 left associative and p4 right associative, and p5 left associative and p5 right associative.

The results with the different settings on Otter were similar; all the settings found the proofs of the same problems in a short amount of time and timed out on the rest of the problems. So only one set of results on Otter is reported, those obtained using negative hyper-resolution and only the negation of the theorem in the set of support. Performance of DCTP was similar to that of Otter. Vampire obtained a few more proofs than did Otter, and E-SETHEO obtained some more proofs than Vampire. OSHL-U obtained all the proofs, though on some of the problems that were also proved by other provers, OSHL-U used more CPU time. The “nostats” entries in the Tables correspond to when Vampire timed out but did not output any statistics of progress.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
2	0.175	41	36	600+	100303	24712	0.00	103	90	0.0	0.01
3	0.678	85	80	600+	66753	31496	70.10	3606742	50382	0.3	300+
4	2.107	141	136	600+	47219	22119	300+	25898955	68385	0.3	300+
5	5.317	207	202	600+	46054	20941	300+	25298293	67864	2.6	300+
6	12.019	283	278	600+	60247	22923	300+	25612105	68457	300+	300+
7	38.973	525	521	600+	56299	19660	300+	25641650	67977	300+	300+
8	77.941	663	659	600+	53652	18932	300+	25863117	68542	300+	300+

Table 8.1: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p1 left associative), $S1 \cup S2 \cup \dots \cup Sn = Sn \cup S(n-1) \cup \dots \cup S1$, with both sides associated to the left, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
3	0.718	128	123	600+	67489	31366	15.0	111454	19427	0.2	24.94
4	2.593	280	275	600+	47835	22418	250+	nostats	nostats	0.3	300+
5	8.155	511	506	600+	47291	21539	300+	25888516	68531	6.5	300+
6	22.835	834	829	600+	58489	21778	300+	25697427	68648	300+	300+
7	82.979	1380	1376	600+	57085	19474	300+	25882741	68756	300+	300+
8	164.227	2086	2082	600+	55813	18944	300+	25939919	68790	300+	300+

Table 8.2: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p1 right associative), $S1 \cup S2 \cup \dots \cup Sn = Sn \cup S(n-1) \cup \dots \cup S1$, with the left side associated to the left and the right side associated to the right, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
1	0.072	13	8	0.01	62	30	0.0	52	45	0.0	0.01
2	0.402	59	54	600+	46693	21377	20.0	146848	21002	0.0	300+
3	1.829	164	159	600+	51421	24494	300+	26129081	68237	0.0	300+
4	6.86	340	335	600+	41275	19441	300+	25329011	68114	3.0	300+
5	21.720	600	595	600+	41347	19075	300+	25647655	68034	300+	300+
6	100.499	1161	1159	600+	51921	19852	300+	25835826	68478	300+	300+
7	207.153	1775	1771	600+	51565	18214	300+	nostats	nostats	300+	300+

Table 8.3: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p2 left associative), $S1 \cup S2 \cup \dots \cup Sn = S1 \cup S2 \cup \dots \cup Sn \cup S1 \cup S2 \cup \dots \cup Sn$ with both sides associated to the left, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+(300+) means that a proof was not found in 600(300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
2	0.414	53	48	600+	46565	21553	20.0	184651	23354	0.0	300+
3	3.227	200	195	600+	53923	25718	300+	26095006	68624	35.0	300+
4	8.474	335	330	600+	44199	20624	300+	nostats	nostats	300+	300+
5	40.012	598	596	600+	44254	20050	300+	25440159	67256	300+	300+
6	112.260	1008	1004	600+	58893	22325	300+	nostats	nostats	300+	300+
7	180.490	1279	1275	600+	56941	19866	300+	nostats	nostats	300+	300+

Table 8.4: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p2 right associative), $S1 \cup S2 \cup \dots \cup Sn = S1 \cup S2 \cup \dots \cup Sn \cup S1 \cup S2 \cup \dots \cup Sn$, with the left side associated to the left and the right side associated to the right, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
2	0.043	6	3	0.01	9	5	0.0	29	25	0.0	0.0
3	0.876	105	100	600+	67489	31366	65.0	2997523	47238	0.0	300+
4	2.589	176	171	600+	47835	22418	300+	nostats	nostats	0.0	300+
5	6.357	258	253	600+	46662	21228	300+	nostats	nostats	0.0	300+
6	24.061	330	326	600+	60927	23223	300+	nostats	nostats	300+	300+
7	35.338	436	432	600+	58891	20602	300+	nostats	nostats	300+	300+
8	58.917	553	549	600+	58237	20342	300+	nostats	nostats	300+	300+

Table 8.5: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p3), $S1 \cup S2 \cup \dots \cup Sn = S1 \cup S2 \cup \dots \cup Sn$ with the left side associated to the left and the right side associated to the right, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
2	0.123	17	12	600+	135461	8533	0.0	72	63	0.0	0.02
3	0.462	75	70	600+	125759	8786	40.1	227258	46580	0.0	300+
4	1.432	131	126	600+	98812	6108	300+	27382397	72420	0.0	300+
5	3.584	197	192	600+	97230	6124	300+	27042468	72461	0.0	300+
6	7.752	273	268	600+	93064	5881	300+	nostats	nostats	110.1	300+
7	15.026	359	354	600+	90592	5741	300+	27478904	72790	300+	300+
8	45.355	635	631	600+	82116	5254	300+	nostats	nostats	300+	300+

Table 8.6: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p4 left associative), $S1 \cap S2 \cap \dots \cap Sn = Sn \cap S(n-1) \cap \dots \cap S1$ with both sides associated to the left, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
3	0.704	97	92	600+	108841	8266	0.0	3504	3141	0.0	300+
4	2.424	258	253	600+	93254	5853	25.0	119409	37380	0.0	300+
5	7.458	524	519	600+	91392	5751	300+	nostats	nostats	0.0	300+
6	20.826	919	914	600+	89726	5672	300+	nostats	nostats	0.0	300+
7	52.417	1466	1461	600+	87254	5532	300+	27360321	72279	2.3	300+
8	132.051	1712	1708	600+	85270	5420	300+	nostats	nostats	300+	300+

Table 8.7: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p4 right associative), $S1 \cap S2 \cap \dots \cap Sn = Sn \cap S(n-1) \cap \dots \cap S1$ with the left side associated to the left and the right side associated to the right, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
1	0.071	13	8	0.01	55	37	0.0	53	46	0.0	0.01
2	0.435	38	33	600+	154993	22287	5.0	20420	17774	0.0	300+
3	2.034	185	180	600+	98752	6385	300+	27402260	72208	0.0	300+
4	6.728	409	404	600+	97008	6100	300+	nostats	nostats	0.0	300+
5	20.013	756	751	600+	94922	5985	300+	27301184	72478	0.0	300+
6	53.247	1252	1247	600+	90714	5729	300+	26978451	72582	300+	300+
7	193.93	1301	1297	600+	86744	5498	300+	27462355	72280	300+	300+

Table 8.8: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p5 left associative), $S1 \cap S2 \cap \dots \cap Sn = S1 \cap S2 \cap \dots \cap Sn \cap S1 \cap S2 \cap \dots \cap Sn$ with both sides associated to the left, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+(300+) means that a proof was not found in 600(300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
2	0.416	64	59	600+	155839	22549	0.0	2228	2042	0.0	300+
3	2.974	233	228	600+	88957	5900	300+	27302710	72327	0.0	300+
4	6.906	370	365	600+	87594	5510	300+	nostats	nostats	0.0	300+
5	32.901	518	514	600+	85472	5393	300+	27577403	72123	4.9	300+
6	67.015	860	856	600+	82570	5254	300+	27366758	72571	300+	300+
7	85.133	1106	1102	600+	79408	5110	300+	nostats	nostats	300+	300+

Table 8.9: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p5 right associative), $S1 \cap S2 \cap \dots \cap Sn = S1 \cap S2 \cap \dots \cap Sn \cap S1 \cap S2 \cap \dots \cap Sn$ with the left side associated to the left and the right side associated to the right, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

n	OSHL			Otter			Vampire			E	D
	time sec.	clauses		time sec.	clauses		time sec.	clauses		time sec.	time sec.
		gen.	kept		gen.	kept		gen.	kept		
2	0.040	6	3	0.01	8	5	0.0	29	25	0.0	0.0
3	0.621	104	99	600+	108753	7731	300+	27355902	70805	0.0	300+
4	1.803	177	172	600+	91948	5688	300+	27362158	71521	0.0	300+
5	4.300	263	258	600+	91970	5792	300+	27366993	72035	4.9	300+
6	8.858	360	355	600+	88480	5595	300+	27397879	71703	10.0	300+
7	23.639	431	427	600+	85804	5447	300+	nostats	nostats	300+	300+
8	36.909	546	542	600+	83922	5344	300+	nostats	nostats	300+	300+

Table 8.10: Timing and clauses of OSHL, Otter, Vampire, E-SETHEO and DCTP on set of theorems (p6), $S1 \cap S2 \cap \dots \cap Sn = S1 \cap S2 \cap \dots \cap Sn$ with the left side associated to the left and the right side associated to the right, for various values of n . Time is in seconds. E denotes E-SETHEO, D denotes DCTP. 600+ (300+) means that a proof was not found in 600 (300) seconds.

8.4 Discussion

The power of OSHL-U here is largely due to the case analysis rule. Definitions are frequently of the form $L == A$ where L is a literal and A is a first-order formula. In clauses obtained from such definitions, it is frequently the case that one of the literals (the literal that comes from L) has all the variables in the clause. This will be true if A has no quantifiers in it, for example. This is true for all of the clauses in the axioms used here, except one of the subset axioms. And when one literal in a clause C has all the variables, then the clause C can be used by the case analysis rule. This explains why the case analysis rule helps for definitions. As for the remaining subset axiom (in which no literal has all the variables), this axiom can frequently be handled by the ground UR resolution rule or the unit filtering rule.

The rule analyses the literals of a clause individually, thus essentially performing definition expansion on each literal, in turn. For example, the definition of set equality in terms of subset is expressed by the clause $\{X=Y, \neg X \subset Y, \neg Y \subset X\}$. When trying to prove equality of two sets, the literal $X=Y$ matches to a literal from the theorem, so an instance of the clause gets selected by the U rules. Subsequently, instances matching the literals $\neg X \subset Y$ and $\neg Y \subset X$ get selected by the U rules. Propositional unsatisfiability of the set of instances is detected by a propositional decision procedure and the original clauses are still available for inferences, so directionality of the definitions is not a concern. The U rules simulate the replacement method described in detail in the earlier work on OSHL with replacement rules [PZ99]. That paper also explains in more detail why the U rules work so well for definitions, though the terminology used there is different. However, the earlier method is not complete, while the current OSHL-U is a complete theorem proving strategy.

The size-lexicographic ordering on ground clauses used by OSHL-U causes instances of smaller size to be generated earlier. However, the heuristic favoring ground terms in the theorem by giving these terms a size measure similar to a constant term allows early generation of instances with large sizes containing large terms that appear in the theorem. This further helps on our test problems that contain large terms in the theorem.

The reason Otter does so poorly on these problems is that they are highly non-Horn and resolution is slow on them. Also, the terms generated are large and Otter generates clauses in order of size, so it takes a long time for Otter to get to clauses that are big enough for the proofs.

8.5 Conclusions

OSHL-U does well on definitions compared to Otter and Vampire and compared to resolution in general. OSHL-U also has better performance on these problems than E-SETHEO and DCTP. Several different settings were tried in Otter using a single uniform strategy each time. Vampire and E-SETHEO used strategy scheduling, whereby a set of possible good strategies are chosen based on the problem, and these strategies are scheduled to be attempted in some order till a solution can be found. The TPTP contains several hundreds of problems in the domain of Set Theory (SET), which contain axioms of set theory. As seen in Chapter 4, OSHL-U has demonstrated superior performance over Otter, in terms of number of proofs obtained with a fixed time limit, on these problems as well. This further supports our observation about the superiority of OSHL-U performance on problems involving definition expansion.

Chapter 9

Semantic Guidance in Proof Search

Semantics refers to the meaning of symbols occurring in a problem. A first-order logic description of a theorem proving problem gives us a syntactic representation of the problem. However, the syntactic symbols occurring in the problem also correspond to concepts in the problem domain. For example, predicates such as 'move' or 'grab' that may be used describe problems in robotic planning have a corresponding meaning in the physical world; predicates such as 'subset' and 'member' that may be used to describe problems in set theory correspond to certain mathematical concepts. Such problem-specific information, or semantics, often helps a human reason about the problem. Enabling a theorem prover to effectively use semantics in its proof search continues to be one of the biggest challenges in automated deduction.

In the early days of automated deduction research, semantics were successfully used to guide proof search in the Geometry Theorem Proving Machine developed by Gelernter at IBM [GHL63]; the Geometry Theorem Proving Machine used a geometry diagram as semantic information to prune irrelevant candidates from the search tree to prove many results from high-school geometry. Many current theorem provers are based primarily on the hyper-resolution strategy. Hyper-resolution performs several resolution inference steps at a time, and therefore makes larger and fewer inference steps than does binary resolution. However, it blindly manipulates symbols and has no concept of the meanings of the symbols. Slagle showed hyper-resolution to be a special case of *semantic resolution* [Sla67]. Semantic resolution uses truth valuation of the syntactic symbols, i.e., a model, to guide inference steps. Hyper-resolution uses a static trivial model, which evaluates everything to true or everything to false, and is therefore purely syntactic in operation. John Slaney developed algorithms to use changing models to guide resolution inferences, referred to as dynamic semantic resolution. There is some evidence that dynamic semantic resolution has the potential to outperform hyper-resolution [Tho03].

Ganzinger, Meyer and Weidenbach developed a method [GMW97], that is a variant of ordered resolution with semantic restrictions and is very similar to OSHL. Both this method and OSHL specify a model of a subset of input clauses, which is minimal in a certain ordering on interpretations, and try to find clauses that contradict the model. OSHL generates an instance that contradicts the model and modifies the model using this instance; two ground clauses may or may not be resolved. However, the method of Ganzinger et al resolves two clauses, one of which is a minimal clause contradicting the model. Also, OSHL allows user-specified semantics, whereas Ganzinger et al use a fixed initial interpretation which is minimal in a certain ordering on interpretations. This latter difference makes OSHL goal-sensitive, which Ganzinger et al is not. In addition, Ganzinger et al works at the first-order level, while OSHL extends the propositional method to first-order logic by an enumeration method that can introduce inefficiencies.

9.1 Use of Semantics in OSHL-U

The power of the OSHL design comes from the ability of the user to supply a model to guide instance generation in proof search. The models provide the prover with semantic information specific to the problem. So, in the context of OSHL, semantics refers to the model or interpretation supplied at the start of proof search. The OSHL algorithm was originally intended to be used only in conjunction with semantic guidance to avoid blind enumeration. This accounts for the relatively poor performance of the OSHL strategy using trivial semantics and without the use of U rules as reported in Chapter 4. All the results reported with OSHL-U so far have also used trivial semantic models that interpret every predicate to true or every predicate to false, hence performing purely syntactic inference steps. Some of the theoretical results of Plaisted have shown that OSHL, when used with appropriate semantics, implicitly performs unifications. Thus the choice of appropriate semantics should profoundly affect OSHL performance. In general, natural semantics – that is, a semantics that corresponds to the mathematical or physical meaning of the symbols and satisfies all the axioms – could help the proof search by providing to the automated theorem prover the same kind of information that is available to a human prover. Standard semantics are known for many domains and it seems reasonable to allow the prover to take advantage of this knowledge. OSHL-U is capable of using user-specified semantic models in the same way as OSHL and this chapter demonstrates that semantics can indeed be used to better guide instance generation. We measure the search space, i.e., the number of clauses generated and execution times to show that the performance of OSHL-U can be improved with the use of non-trivial semantics.

9.1.1 An Example: “Who Killed Aunt Agatha?”

We give an example of the use of semantics on the problem, PUZ001-2, from the PUZ (puzzles) domain of the TPTP. The logic puzzle, stated in English, is as follows:

Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadbury Mansion, and are the only people who live therein. A killer always hates his victim, and is never richer than his victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone Aunt Agatha hates. No one hates everyone. Agatha is not the butler. Therefore, Agatha killed herself.

The input consists of 26 clauses and 53 literals. The predicates in the problem are *equal*(X, Y), *lives_at_dreadbury*(X), *hates*(X, Y), *richer*(X, Y) and *killed*(X, Y). The constants are *aunt_agatha*, *butler*, *charles* and *someone*; *everyone_but*(X) is a Skolem function that arises from the statement – No one hates everyone.

A human is able to reason about the puzzle and solve it, even without the use of formal logic. But if one were to present the same problem, replacing all the predicates, functions, and constants with names such a *pred*₁, *pred*₂, *func*₁, and so on, then it would become a lot more difficult for the human to solve. This is because a human is able to interpret the semantics of the problem in a certain way, which helps her solve the problem. For example, a human “knows” that there are only 3 persons and one of those 3 persons is the killer. This means that the domain of definition to consider in solving this problem should have 3 elements. A human also “knows” that a person can not be richer than himself/herself, and interpretes the *richer*(X, Y) predicate accordingly. However, with a purely syntactic formulation of the problem, such additional semantic information is lost, making it harder to solve the problem. Because an automated theorem prover lacks this kind of human “knowledge” of the problem, a human user can supply this extra information to the prover through a semantic model. The semantic model provides the prover with an initial interpretation that incorporates extra information that the human user knows about the problem.

In order to solve this problem, we supplied OSHL-U with a non-trivial semantic model consisting of 3 elements. We did not provide a natural semantics. Some of the input clauses are not modeled by our initial semantics, which maps the *killed* predicate such that Charles is the killer. Also, it may not be apparent to a human user how the function *everyone_but* (arising from Skolemization) should be interpreted. We mapped the function to be consistent

with the mapping for the *hates* predicate. The domain of definition $D = \{1, 2, 3\}$. The mappings of the constants, function, and predicates are as follows.

aunt_agatha \mapsto 1
 butler \mapsto 2
 charles \mapsto 3
 someone \mapsto 1

everyone_but : $D \mapsto D$
 1 \mapsto 2
 2 \mapsto 2
 3 \mapsto 3

equal : $D \times D \mapsto \{\text{True}, \text{False}\}$
 $(X, Y) \mapsto \text{True}$, if $X = Y$
 $(X, Y) \mapsto \text{False}$, otherwise

lives_at_dreadbury : $D \mapsto \{\text{True}, \text{False}\}$
 1 \mapsto True
 2 \mapsto True
 3 \mapsto True

hates : $D \times D \mapsto \{\text{True}, \text{False}\}$
 (1, 1) \mapsto True
 (1, 2) \mapsto False
 (1, 3) \mapsto True
 (2, 1) \mapsto True
 (2, 2) \mapsto False
 (2, 3) \mapsto True
 (3, 1) \mapsto False
 (3, 2) \mapsto True
 (3, 3) \mapsto False

richer : $D \times D \mapsto \{\text{True}, \text{False}\}$
 (1, 1) \mapsto False
 (1, 2) \mapsto False
 (1, 3) \mapsto True
 (2, 1) \mapsto True
 (2, 2) \mapsto False

$(2, 3) \mapsto \text{True}$
 $(3, 1) \mapsto \text{False}$
 $(3, 2) \mapsto \text{False}$
 $(3, 3) \mapsto \text{False}$

$\text{killed} : D \times D \mapsto \{\text{True}, \text{False}\}$
 $(X, Y) \mapsto \text{True}, \text{ if } X = 3 \text{ and } Y = 1.$
 $\text{False}, \text{ otherwise}$

With the non-trivial semantics described, OSHL-U finds the proof in 5 minutes 49 seconds, generating 1795 clauses. On changing the described semantics to model Aunt Agatha as the killer, OSHL-U finds the proof in 5 minutes 48 seconds, generating 1632 clauses. On changing the described semantics by using different mappings for the *hates* and the *richer* predicates, OSHL-U still obtains the proof in about 6 minutes. With an all-positive or all-negative semantics, OSHL-U runs for over 3 hours generating more than 50,000 clauses, without finding a proof. Therefore, in this case, a user-specified semantics produces significant improvement in OSHL-U performance over a trivial semantics.

9.1.2 Semantics in Group Theory Problems

We also tested the use of user-specified semantics on some group theory (GRP) problems. Table 9.1 shows the results obtained with OSHL-U using a non-trivial semantics compared to those with OSHL-U using trivial semantics and with Otter in the “auto” mode, on these problems. Cases when a prover timed out without generating a proof are marked with “fail” and the execution time allotted to the proof attempt is noted.

In mathematics, a group is a set, with a binary operation on elements of the set, such as multiplication or addition, satisfying certain axioms. We used a non-trivial natural semantics that models a finite group of size 4. The domain elements were mapped to the integers 0, 1, 2, 3 and the binary operation was addition modulo 4. The semantics chosen was suitable for all the problems in Table 9.1 except GRP008-1. These problems are theorems about identity and inverse functions in a group and have only Horn clauses; problem GRP008-1 is stated to be a theorem of “unknown meaning” and has 1 non-Horn clause. On some of the problems, use of this semantics gives the proof faster and with the generation of fewer clauses than using either of the trivial semantics. Use of the semantics also helps to obtain proofs of some problems that could not be proved with trivial semantics. Problems in GRP are mostly all Horn, so Otter exhibits good performance on these problems. However, there are a few

Problem	Natural Sem.		All-positive		All-negative		Otter	
	gen	time(sec)	gen	time(sec)	gen	time(sec)	gen	time(sec)
GRP003-1	140	119.20	fail	300+	fail	300+	116	0.01
GRP004-1	53	28.00	fail	300+	fail	300+	129	0.00
GRP004-2	222	716.50	fail	300+	fail	300+	335	0.01
GRP007-1	17	0.38	18	1.99	58	3.8	85	0.01
GRP008-1	396	226.30	fail	600+	fail	600+	fail	600+
GRP017-1	241	16.05	fail	300+	fail	300+	210	0.02
GRP018-1	15	0.48	36	6.40	108	6.14	266	0.01
GRP019-1	14	0.24	39	7.90	fail	300+	267	0.01
GRP020-1	20	1.55	68	33.80	fail	300+	265	0.02
GRP021-1	18	0.87	45	5.55	fail	300+	264	0.01
GRP022-1	36	17.90	fail	600+	fail	600+	448	0.02
GRP023-1	16	0.50	15	0.33	fail	300+	79	0.01
GRP023-2	36	1.91	23	0.69	fail	300+	fail	300+

Table 9.1: Execution time and number of clauses generated with OSHL-U and a non-trivial natural semantics, with OSHL-U and trivial semantics (all-positive and all-negative), and with Otter in the “auto” mode. The number of clauses generated and the execution time in seconds are shown. 300+ (600+) means that the prover timed out in 300 (600) seconds without finding a proof.

problems proved by the semantics that even Otter, in the autonomous mode, could not prove.

We also tested non-natural semantics on some GRP problems. In these cases, the groups were sets of integers of size 2 ($\{0,1\}$) and 4 ($\{0,1,2,3\}$). However, the binary operation was selected to be such that the axioms of group theory are not all satisfied by the model. Table 9.2 shows the result of these tests. These indicate that even a non-natural semantics can perform better than trivial semantics on some problems.

Problem	Non-natural Sem.			All-positive		All-negative		Otter	
	size	gen	time(sec)	gen	time(sec)	gen	time(sec)	gen	time(sec)
GRP005-1	2	6	0.02	6	0.02	6	0.02	57	0.02
GRP008-1	2	90	16.6	fail	600+	fail	600+	fail	600+
GRP018-1	2	21	0.97	36	6.40	108	6.14	266	0.01
GRP019-1	2	22	1.570	39	7.920	fail	300+	267	0.01
GRP034-3	4	25	2.242	44	4.183	84	10.185	141	0.01

Table 9.2: Execution time and number of clauses generated with OSHL-U and a non-trivial non-natural semantics, with OSHL-U and trivial semantics (all-positive and all-negative), and with Otter in the “auto” mode. The domain size is given for the non-trivial semantics. The number of clauses generated and the execution time in seconds are shown. 300+ (600+) means that the prover timed out in 300 (600) seconds without finding a proof.

We performed the experiments with natural semantics of larger sizes using groups of sizes 16, 24, and 40. The results were similar to those with groups of size 4. Using larger semantics, proofs of the same problems were found as with semantics of size 4 generating the same number of clauses in similar execution times. Adding more elements did not increase the information conveyed by the semantics; proofs of the same problem were not found any faster and no new proofs were found. At the same time, an increase in the number of terms did not result in more clauses being generated.

Chapter 10

Conclusion

This dissertation explored propositional techniques to first-order theorem proving with the implementation of the OSHL-U theorem prover. The OSHL-U prover builds on and enhances the OSHL theorem prover, which is based on propositional techniques applied to first-order logic. We showed that a propositional-style prover like OSHL-U can come within the range of performance of respectable resolution theorem provers and even outperforms resolution provers on some classes of problems. These classes of problems are difficult for resolution provers; so by providing alternative techniques for such problems, we increase the power of first-order theorem provers, in general.

We presented U rules that extend the OSHL strategy; the resulting theorem prover was implemented and is referred to as OSHL-U. The U rules provide syntactic guidance in proof search to the OSHL algorithm, and give tremendous speedup in performance. OSHL-U was tested on 4417 problems from the TPTP v2.5.0. In the absence of sophisticated semantic guidance and within 30 seconds of execution time on each problem, OSHL-U obtains 238 proofs without the U rules, 900 proofs with the U rules, and 1027 proofs with the U rules and a more optimized implementation. The problems were run without sophisticated semantics because generating semantics on individual problems or problem domains requires extensive user input and, therefore, not feasible to perform automatically in a short amount of time on a large collection of problems. The 30 second time limit was chosen to enable us to collect a set of results on all the problems within a few days.

Otter, a resolution prover, was also tested on the TPTP problems and in the autonomous mode, Otter obtained 1697 proofs within 30 seconds on each problem. It is interesting that OSHL-U, a propositional prover, can obtain more than half the number of the proofs that Otter obtains in the autonomous mode on the TPTP problems in 30 seconds. This is especially so when one considers that the propositional prover has no special rules for equality axioms and has a far less sophisticated implementation than Otter. OSHL and prior propositional

provers have performed well on near-propositional problems. This is the first time to our knowledge that a propositional style prover, not performing unification on non-propositional literals, has demonstrated performance comparable to that of a resolution prover on a group of TPTP problems that is not near-propositional in structure. This suggests that propositional techniques have the potential to speed up a significant number of first-order proofs. OSHL-U outperforms Otter on SET and FLD problems which have many non-Horn clauses, indicating propositional techniques are superior to resolution on such problems.

We looked at space efficiency and inference efficiency of OSHL-U and Otter. In search space as well as storage space, OSHL-U is more efficient than Otter on non-Horn problems, and Otter is more efficient than OSHL-U on Horn problems. This observation also holds for inference efficiency.

An interesting question in theorem proving is how essential unification of non-ground literals is to theorem proving, and how much theorem proving one can do without it. Resolution is efficient if the problem is UR resolvable, meaning that the proof can be found by UR resolution. All Horn problems are UR resolvable, and some non-Horn problems are, too. In the early years of theorem proving, the problems people looked at belonged to these classes, which were easy for resolution. This, therefore, gave resolution an advantage. Some theoretical results of Plaisted have shown that with a high probability, hard problems are not Horn or UR resolvable. This was also confirmed by our experimental results with Otter that have shown, that of the 1697 TPTP problems provable by Otter in 30 seconds, at least 1042 can be proved by UR resolution; of the 297 problems proved by Otter that are both non-Horn and had TPTP rating greater than zero, at most 215 are not UR resolvable. Therefore, resolution tends to be inefficient on hard problems that are not UR resolvable. The performance of OSHL-U suggests propositional techniques are superior to resolution on non-Horn problems which are not UR resolvable and are, therefore, difficult for resolution provers. Thus OSHL-U and propositional provers significantly enhance the power of automated theorem proving.

OSHL with replacement rules for definition handling has done well on problems involving definition expansion. However, OSHL-U is a general theorem prover and does not have special rules for definitions. Nevertheless, OSHL-U exhibits better performance than Otter and several other leading theorem provers - Vampire, E-SETHEO and DCTP - on some classes of problems that involve definition expansion for proofs.

OSHL-U was also tested with non-trivial user-specified semantics on some GRP problems. The results of these tests show that OSHL-U generates fewer clauses and uses less execution time in obtaining proofs with such semantics, even in cases where natural semantics were not used, compared to OSHL-U with trivial semantics.

10.1 Future Work

The OSHL-U implementation is rather naive. There is no data structure support for storing or matching literals or terms. Ocaml List and Ocaml Set and the operations defined on these have been used in the implementation. OSHL-U also has no special rules for term-rewriting or handling equality axioms; its performance might be significantly enhanced by term-rewriting, special rules for equality axioms, and better data structures.

The unit filter rule of inference in OSHL-U provides a mechanism to interface OSHL-U with another theorem prover. Thus, the resolvents from a conventional theorem prover can be added to the set of clauses used by OSHL-U for filtering. Interesting experiments can be conducted using OSHL-U in conjunction with an efficient resolution prover.

Use of domain-specific semantic information to guide proof search has long been believed to be a potential area of improving automated reasoning techniques. However, till date, this remains a largely unrealized dream. OSHL and OSHL-U are capable of using semantic models. Such models could be supplied by a human expert or generated automatically. Machine-generation of models could probably benefit from heuristics. Experiments with machine-generated models might help develop such heuristics and help reveal interesting facts about selection of appropriate semantics for a problem.

BIBLIOGRAPHY

- [AB03] Martín Abadi and Bruno Blanchet. Computer-Assisted Verification of a Protocol for Certified Email. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium (SAS'03)*, volume 2694 of *Lecture Notes on Computer Science*, pages 316–335, San Diego, California, June 2003. Springer Verlag.
- [AJK⁺00] Mark D. Aagaard, Robert B. Jones, Roope Kaivola, Katherine R. Kohatsu, and Carl-Johan H. Seger. Formal verification of iterative algorithms in microprocessors. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 201–206, New York, NY, USA, 2000. ACM Press.
- [AP92] Geoffrey D. Alexander and David A. Plaisted. Proving equality theorems with hyperlinking. In Deepak Kapur, editor, *CADE-12 – The 12th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 706–710. Springer, 1992.
- [BA98] Matthew Bishop and Peter B. Andrews. Selectively instantiating definitions. In *CADE-15: Proceedings of the 15th International Conference on Automated Deduction*, pages 365–380, London, UK, 1998. Springer-Verlag.
- [Bau00] Peter Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831, pages 200–219. Springer, 2000.
- [BF91] W. W. Bledsoe and G. Feng. Set-Var. *Journal of Automated Reasoning*, 11:293–314, 1991.
- [BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001.
- [Bil96] Jean-Paul Billon. The disconnection method - a confluent integration of unification in the analytic framework. In *TABLEAUX '96: Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 110–126, London, UK, 1996. Springer-Verlag.
- [Bla03] Bruno Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *Dagstuhl seminar "Language-Based Security"*, October 2003.
- [Ble97] W. W. Bledsoe. Non-resolution theorem proving. In *Artificial Intelligence*, 9, pages 1–35, 1997.
- [BT03] P. Baumgartner and C. Tinelli. The model evolution calculus, 2003.
- [CB98] Yirn-An Chen and Randal E. Bryant. Verification of floating-point adders. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 488–499, London, UK, 1998. Springer-Verlag.

- [CCH⁺96] Yirng-An Chen, Edmund M. Clarke, Pei-Hsin Ho, Yatin Vasant Hoskote, Timothy Kam, Manpreet Khaira, John W. O’Leary, and Xudong Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In *FMCAD ’96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 19–33, London, UK, 1996. Springer-Verlag.
- [CG01] Shang-Ching Chou and Xiao-Shan Gao. Automated reasoning in geometry. In *Handbook of Automated Reasoning*, pages 707–749. 2001.
- [CGZ96] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. An introduction to geometry expert. In *CADE*, pages 235–239, 1996.
- [CHM02] K. Claessen, R. Hahnle, and J. Martensson. Verification of hardware systems with first-order logic, 2002.
- [CL71] C.-L. Chang and R. C.-T. Lee. Herbrand’s theorem. In C.-L. Chang and R. C.-T. Lee, editors, *Symbolic Logic and Mechanical Theorem Proving*, pages 45–69. Academic Press, New York, 1971.
- [DA95] D.P. Appenzeller and A. Kuehlmann. Formal Verification of a PowerPC Microprocessor. In *Proceedings of the IEEE International Conference on Computer Design (ICCD ’95)*, Austin, Texas, October 1995. IBM.
- [Dav83] M. Davis. The prehistory and early history of automated deduction. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pages 1–28. Springer, Berlin, Heidelberg, 1983.
- [DHK98] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo, 1998. Technical Report 3400, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay, France.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *CACM*, 5:394–397, 1962.
- [DM96] D. Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 135–146, New Brunswick, NJ, USA, / 1996. Springer Verlag.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [DP03] S. Das and D. A. Plaisted. An improved propositional approach to first-order theorem proving. In *Workshop on Model Computation - Principles, Algorithms, Applications at The 19th International Conference on Automated Deduction*, 2003.
- [Fit96] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
- [Fun01] Shelby Funk. Using a modified size measure to guide the search in the ordered semantic hyper-linking theorem prover. Available at <http://www.cs.unc.edu/shelby/MathResearch.html>, 2001.

- [GHL63] H. Gelernter, J.R. Hansen, and D.W. Loveland. Empirical explorations of the geometry theorem proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 153–167. McGraw-Hill, New York, 1963.
- [Gil60] P. C. Gilmore. A proof method for quantification theory. *IBM Journal of Research and Development*, 4:28–35, 1960.
- [GK03] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, pages 55–64. IEEE Computer Society Press, 2003.
- [GMW97] H. Ganzinger, Chr. Meyer, and Chr. Weidenbach. Soft typing for ordered resolution. In *Automated Deduction — CADE’14*, volume 1249 of *Lecture Notes in Computer Science*, pages 321–335, Berlin, 1997. Springer-Verlag.
- [GOP93] D. Gabbay, J. Ohlbach, and D. Plaisted. Killer transformations. In *1993 Workshop on Proof Theory in Modal Logic*, pages 1–45, 1993.
- [HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. Waldmeister - Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in *Lecture Notes in Artificial Intelligence*, pages 232–236. Springer-Verlag, 1999.
- [KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Compass’96: Eleventh Annual Conference on Computer Assurance*, pages 23–34, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.
- [LBM98] Helen Lowe, Alan Bundy, and Duncan McLean. The use of proof planning for cooperative theorem proving. *Journal of Symbolic Computation*, 25:239–261, 1998.
- [Lov68] Donald W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM*, 15(2):236–251, 1968.
- [Lov69] D. Loveland. A simplified format for the model elimination procedure. *Journal of the ACM*, 16:349–363, 1969.
- [LP94] S.-J. Lee and D. Plaisted. Use of replace rules in theorem proving. *Methods of Logic in Computer Science*, 1:217–240, 1994.
- [LS01] R. Letz and G. Stenz. DCTP: A Disconnection Calculus Theorem Prover. In *Lecture Notes in Artificial Intelligence*, volume 2083, pages 381–385. Springer Verlag, 2001.
- [LSBB92] Reinhold Letz, Johann Schumann, S. Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.
- [MB88] R. Manthey and F. Bry. SATCHMO: A Theorem Prover Implemented in Prolog. In *Proceedings of the Ninth International Conference on Automated Deduction*, *Lecture Notes in Computer Science* 310, pages 415–434. Springer-Verlag, 1988.
- [McC90] W. McCune. Otter 2.0 (theorem prover). In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 663–4, July 1990.

- [McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, Illinois, 1994.
- [McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.
- [McC03] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MS-C-TM-263, Argonne National Laboratory, Argonne, USA, 2003.
- [MP05] S. Miller and D. A. Plaisted. The space efficiency of OSHL. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, 2005.
- [MRS01] D. McMath, M. Rozenfeld, and R. Sommer. A computer environment for writing ordinary mathematical proofs. In *LPAR - 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), pages 507–516. Springer-Verlag, 2001.
- [MS98] Padraic Monaghan and Keith Stenning. Learning to solve syllogisms by watching others’ learning, 1998. Online Research Papers of the Human Communication Research Center, University of Edinburgh.
- [NSS56] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Western Joint Computer Conference*, pages 218–239, 1956.
- [NSS63] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations of the geometry theorem proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 134–152. McGraw-Hill, New York, 1963.
- [NSS83] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pages 49–73. Springer, Berlin, Heidelberg, 1983.
- [Par97] M. Paramasivam. Instance-based first-order methods using propositional calculus provers, 1997. PhD thesis, University of North Carolina at Chapel Hill.
- [PC94] D. Plaisted and Heng. Chu. Semantically guided first-order theorem proving using hyper-linking. In *Twelfth International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence 814, pages 192–206, 1994.
- [PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.
- [PL92] D. A. Plaisted and Shie-Jue Lee. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.
- [Pla94] D. A. Plaisted. Ordered semantic hyper-linking. Technical Report MPI-I-94-235, Max-Planck Institut fuer Informatik, Saarbruecken, Germany, 1994.

- [Pla99] David A. Plaisted. Theorem proving. 21:662–682, 1999.
- [PP91] David A. Plaisted and Richard C. Potter. Term rewriting: Some experimental results. *J. Symb. Comput.*, 11(1/2):149–180, 1991.
- [PP97] M. Paramasivam and David A. Plaisted. Rrtp - a replacement rule theorem prover. *J. Autom. Reasoning*, 18(2):221–226, 1997.
- [PP98] M. Paramasivam and David A. Plaisted. Automated deduction techniques for classification in description logic systems. *Journal of Automated Reasoning*, 20(3):337–364, 1998.
- [PZ99] D. A. Plaisted and Y. Zhu. Replacement rules with definition detection. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, Lecture Notes in Artificial Intelligence 1761, pages 80–94, 1999. Invited paper.
- [PZ00] D. A. Plaisted and Y. Zhu. Ordered semantic hyper linking. *Journal of Automated Reasoning*, 25(3):167–217, October 2000.
- [Qua92] A. Quaife. Automated deduction in NBG set theory. *Journal of Automated Reasoning*, 8(1):91–147, 1992.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [RV99] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *The 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, 1999.
- [RV02] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
- [Sch02] S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.
- [SK88] F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362. Kaufmann, Los Altos, CA, 1988.
- [Sla67] James R. Slagle. Automatic theorem proving with renamable and semantic resolution. *J. ACM*, 14(4):687–697, 1967.
- [SM84] P. Suppes and J. McDonald. Student use of an interactive theorem prover. In Bledsoe and Loveland, editors, *Automated Theorem Proving: After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 315–360. American Mathematical Society, 1984.
- [Smi90] D. R. Smith. Kids: A semi-automated program development system. In *Special Issue on Formal Methods*, IEEE Transactions on Software Engineering, pages 1024–1043. September 1990.
- [SR95] S. Tahar and R. Kumar. Formal Specification and Verification Techniques for RISC Pipeline Conflicts. *The Computer Journal*, 38(2):111–120, 1995.

- [SS98a] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- [SS98b] Christian B. Suttner and Geoff Sutcliffe. The CADE-14 ATP system competition. *Journal of Automated Reasoning*, 21(1):99–134, 1998.
- [SS99] G. Sutcliffe and D. Seyfang. Smart Selective Competition Parallelism ATP. In A. Kumar and I. Russell, editors, *Proceedings of the 12th Florida Artificial Intelligence Research Symposium*, pages 341–345. AAAI Press, 1999.
- [SW99] G. Stenz and A. Wolf. Strategy Selection by Genetic Programming. In A. Kumar and I. Russell, editors, *Proceedings of the 12th Florida Artificial Intelligence Research Symposium*, pages 346–350. AAAI Press, 1999.
- [SW00] G. Stenz and A. Wolf. E-SETHEO: An Automated Theorem Prover. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-2000)*, number 1847 in Lecture Notes in Artificial Intelligence, pages 436–440. Springer-Verlag, 2000.
- [SWL⁺94] Mark E. Stickel, Richard J. Waldinger, Michael R. Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In *Conference on Automated Deduction*, pages 341–355, 1994.
- [SY97] K. Stenning and P. Yule. Image and language in human reasoning: a syllogistic illustration. In *Cognitive Psychology* 34, 109–159., volume 2, pages 109–159. 1997.
- [Tam97] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.
- [Tam03] Tanel Tammet. Extending classical theorem proving for the semantic web. In *PSSS*, 2003.
- [Tho03] Simon Thornton. Implementing dynamic semantic resolution. In Benjamin I. P. Rubinstein, Nelson Chan, and K. K. Kshetrapalapuram, editors, *Proceedings of the First Australian Undergraduate Students' Computing Conference*, 2003.
- [VB03] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *J. Symb. Comput.*, 35(2):73–106, 2003.
- [Vel04a] Miroslav N. Velev. Exploiting signal unobservability for efficient translation to cnf in formal verification of microprocessors. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10266, Washington, DC, USA, 2004. IEEE Computer Society.
- [Vel04b] Miroslav N. Velev. Using positive equality to prove liveness for pipelined microprocessors. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 316–321, Piscataway, NJ, USA, 2004. IEEE Press.
- [Wei97] C. Weidenbach. SPASS version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, 1997.

- [Wos88] Larry Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall, 1988.
- [ZP97] Yunshan Zhu and D. A. Plaisted. FOLPLAN: A Semantically Guided First-Order Planner. *10th International FLAIRS Conference*, May 1997.