How To Efficiently Implement An OSHL-Based Automatic Theorem Prover

by
Hao Xu

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
2012

Abstract

HAO XU: How To Efficiently Implement An OSHL-Based Automatic Theorem Prover.
(Under the direction of David Plaisted.)

Ordered Semantic Hyper-linking (OSHL) is a general-purpose instance-based first-order automated theorem proving algorithm. Although OSHL has many useful properties, previous implementations of OSHL were not very efficient. The implementation of such a theorem prover differs from other more traditional programs in that a lot of its subroutines are more mathematical than procedural. The low performance of previous implementations prevents us from evaluating how the proof strategy used in OSHL matches up against other theorem proving strategies. This dissertation addresses this problem on three levels. First, an abstract, generalized version genOSHL is defined which captures the essential features of OSHL and for which the soundness and completeness are proved. This gives genOSHL the flexibility to be tweaked while still preserving soundness and completeness. A type inference algorithm is introduced which allows genOSHL to possibly reduce its search space while still preserving the soundness and completeness. Second, incOSHL, a specialized version of genOSHL, which differs from the original OSHL algorithm, is defined by specializing genOSHL. Its soundness of completeness follows from that of genOSHL. Third, an embedded programming language called STACK EL, which allows managing program states and their dependencies on global mutable data, is designed and implemented. STACK EL allows our prover to generate instances incrementally. We also study the performance of our incremental theorem prover that implements incOSHL.

Acknowledgments

I would like to thank my advisor for his guidance and support throughout my Ph.D. study and his help in preparing this dissertation, including reading and commenting on drafts and final version of this dissertation. I would like to thank my Ph.D. committee members for their guidance and support, and reading and commenting on my dissertation. I would like to thank Prof. Maria Paola Bonacina and Prof. Peter Baumgartner for reading and commenting on a draft of this dissertation. I would like to thank everyone at UNC Computer Science, SILS, DICE, and RENCI for creating a wonderful environment for me to complete my Ph.D. study.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

incOSHL is an automated first-order instance-based incremental theorem prover. The algorithm implemented in incOSHL is based on a non-incremental automatic theorem proving (ATP) algorithm called OSHL [33]. The original OSHL algorithm was implemented in Prolog, in a theorem prover which was also called OSHL [33]. An extension of the original algorithm with the U Rules was implemented in Caml, in a theorem prover called OSHL-U [28].[1] The ideas of OSHL have also found their way into a separately developed theorem prover Equinox [11].

OSHL belongs to the larger category of ATP strategies called instance-based methods. The general idea of instance-based methods is: in order to test if a set of logical formulas is satisfiable, we generate instances of it and test if the instances are satisfiable; if they are, then we try to generalize the model for these instances to a general model for the set of logical formulas; otherwise, these instances are a counterexample for the set of logical formulas. This process is analogous to the following problem solving method used in geometry: Given a general, abstract problem, we first draw a geometry diagram and try to figure out how to solve this problem in this particular diagram. The diagram does not represent the general,

[1]The U Rule was shown to accelerate the prover, but they makes the extended algorithm incomplete. As a result, the theorem prover has to fall back to the original algorithm if the extension cannot find a solution.

abstract problem. Rather, it represents an instance of the problem.

There are many instance-based methods [9, 15, 33, 11, 27, 7, 23]. OSHL differs from other instance-based methods in that OSHL works entirely on the ground level. It is an interesting question whether a strategy like that can be efficient.

The evaluation of ATP algorithms has traditionally been based on the performance of particular implementations of those algorithms. The low inference rate (IR) of the previous OSHL-based provers put them at a disadvantage in such evaluations. To compare, at the time they were being developed, they were able to generate instances only on an order of magnitude of 10 clauses per second on commodity hardware [27], while state-of-the-art implementations of resolution-based ATP algorithms were able to generate clauses at a much higher inference rate [34]. What was perhaps encouraging was that despite having significantly lower IR, OSHL-based theorem provers were capable of proving some hard problems [27, 50], which Otter [26] could not prove, in a conventional testing setup. However, in general, the number of problems they could prove (NOPP) in TPTP [44] was lower than that of state-of-the-art theorem provers. This gave rise to the following questions:

1. If the IR of an OSHL-based theorem provers is improved, will its NOPP improve, too?

2. Is there any way to improve the IR of an OSHL-based theorem prover?

3. Is there any way, other than improving the IR, to improve the NOPP for OSHL-based theorem provers?

To answer the first question, we need to either find a way to implement an OSHL-based theorem prover with high IR and test it against problems in TPTP, or simulate such an implementation. As an example of simulation, we can let a slower theorem prover run X times longer than a faster theorem prover, as if the slower prover were implemented in a way such that it was X times more efficient than it actually is. While this second approach might seem attractive at first glance, since it avoids the harder problem of actually improving the IR, there are two potential problems that may put the feasibility of the second approach in question. The first problem is that the IR of the fastest previously implemented OSHL-based theorem prover was thousands of times lower than state-of-the-art theorem provers. The scalar X would be required to be on an order of magnitude of 1000, which is impractical considering the size of TPTP, if we were to run a full simulation. The second, more subtle problem is that letting a theorem prover run X times longer does not necessarily make it generate X times more instances. For example, it may be increasingly more time consuming for a prover to generate new instances as the number of instances generated grows. I will show some evidence of this in Section 8.3. Without alleviating this nonlinear complexity, simply increasing the running time will only reduce the IR. The first approach requires significantly more effort, but in the long run, can be much more beneficial, as once we have an efficient implementation, we can start to explore all kinds of new ideas, and show whether they are significant or not without having to worry about question 1 any more. This leads us to the second question.

When approaching the second question, I took an approach that starts from a more general problem, finds a solution to it, and then applies the solution to our specific application. The more general problem here is to provide a general framework and tools that are not tied to a specific algorithm, to ease the development of instance-based theorem provers, so that, when we want, we can easily experiment with all kinds of new ideas rapidly, while still maintaining relatively high efficiency of the implementation. I will describe in Chapter 6 a low-level framework for implementing ATP algorithms that combines coroutines and dependency of program states on global mutable data. This framework is exposed to programmers in a well-defined, statically-typed embedded language interface called the STACK EL. Our theorem prover is the first theorem prover that I know of which is implemented based on a systematic, language-based approach. As a macro-based embedded language in C/C++, the STACK EL provides some high level constructs that are essential to proof search, with the efficiency closer to that of C/C++. The biggest advantage of this approach is that it permits people to implement provers efficiently with little man power. Perhaps these techniques will be useful for other technical areas as well.

The third question is equally as important because, after all, the performance of the a theorem prover is measured by NOPP, even though IR is an important factor. I will show a term algebra based type inference extension that helps reduce the search space of the OSHL algorithm.[2] One essential feature of type inference is that it tries to discover the semantics that are embedded in the syntax of a problem. For example, if one makes a statement: "The average height of a human is greater than the average height of a tiger," it is clear from the syntax that the following phrase is not relevant: "the average height of the average height of a tiger," since the function "the average height of" is only applied to human or tiger in our statement, and not to some number that is the average height of a tiger. This suggests that we can infer the semantics of symbols from where and how they appear in a problem. This is the basic idea of type inference. Type inference has the potential application in any theorem prover that needs to generate instances by "don't know" non-determinism. Our theorem prover seems to be the first application of type inference to theorem proving that I am aware of in this way.

Before developing the current version of our theorem prover, I developed two

[2]Some people might think it is more accurate to call it sort inference.

previous versions. The first version, written in Java, was called OSHL-S. The objective of developing OSHL-S was:

- develop efficient data structures and subroutines for the original OSHL algorithm;
- develop a caching mechanism for OSHL;
- experiment with type inference.

OSHL-S had an average inference rate on the order of magnitude of 100 clauses per second, which was slightly faster than the previous OSHL implementation. I applied Java profiler to find the hot spot of the prover code for optimization, which helped improve the performance. Bytecode engineering tools were also considered and experimented with for improving the efficiency of the prover. Even though applying bytecode engineering tools could potentially improve the performance of the prover, it would still be limited by the Java virtual machine. On the other hand, the caching mechanism designed was both less adequate and more fragmented than the current version. It was less adequate in that some of the general data structure that were intended to be applied to multiple places were not customizable enough – some glue code had to be written to adapt the general data structure to a specific use. It was fragmented in that in some of the places where the general cache data structure could not be adapted to, a different data structure was written. This created several sets of data structures that had to be maintained separately which increased the overhead for development and maintenance.

With all the bytecode engineering and high complexity of cache data structures, the effort to maintain the code base became higher and higher and it became increasingly difficult to significantly improve the performance of the prover. Eventually, the complexity of maintaining the Java code base seemed to far exceed that of reimplementing the prover from scratch in C/C++. I started to reimplement OSHL-S in C/C++. While designing the architecture of the new prover, I rethought the caching mechanism, a center piece of OSHL-S that had the potential to greatly improve the efficiency of the prover, and divided the program into two strata. The lower stratum, which is implemented as a macro-based embedded language called the STACK EL, deals with general control structures such as program states and activation records. The STACK EL provides a familiar syntax similar to a stand alone program language, by using macros only (hence is portable) and is fully interoperable with C/C++. The higher stratum deals with prover specific subroutines and data structures. The bootstrapping of the development took longer, since I had to develop the STACK EL first. However, once the first version STACK EL was functional, it became much faster to write code for the prover and make changes to the code written. The stratification of the prover code allowed me to develop, maintain, and optimize the two strata of code separately, which greatly improved the efficiency and maintainability of the prover. Also, the caching mechanism was embedded into the STACK EL, giving it more flexibility and avoiding the inadequacy and fragmentation of the caching mechanism in the Java version of OSHL-S. Overall, the C/C++ version of OSHL-S implemented a similar algorithm as OSHL-S. With more efficient data structures and several other improvements, it achieved an average inference rate on the order of magnitude of 1000 clauses per second.

The most important feature of the current version, which is called incOSHL, is that it is capable of generating instances incrementally. incOSHL is based on an improved version of the STACK EL which provides the capability of handling dependency of program states on global mutable data, a crucial feature that enables incremental instance generation, and performs static checking for code written in the embedded language, which helps catch bugs. incOSHL also has a more efficient, region-based memory management mechanism and supports garbage collection. Combining these features, incOSHL achieves a much higher efficiency than OSHL-S. The lessons learned were:

- Java does not provide enough access to low level runtime environment, which is suitable for web development, but not for programs that require extremely high efficiency such as theorem provers;
- choosing the right programming languages for a software project with high complexity is crucial;
- stratification in software design is important for the efficiency of development and maintenance of a complex software project;
- it takes trial and error to come to the right design – it is important to always make sure that the code base is maintainable.

This dissertation focuses on the theoretical and practical aspects of incOSHL. This dissertation is organized as follows:

In the second chapter, we introduce some preliminaries. In the third chapter we introduce genOSHL, a general theoretical framework for designing OSHL-based ATP algorithms, and incOSHL, a sound and complete instance-based ATP algorithm based on a customization of the genOSHL algorithm. In the fourth chapter, we introduce type inference. In the fifth chapter, we introduce a non-incremental implementation of incOSHL. In the sixth chapter, we introduce the STACK EL, a novel low-level algorithm-independent embedded language for implementing ATP algorithms that supports implicit and explicit dependencies of program states on global mutable data. In the seventh chapter, we introduce an incremental implementation of incOSHL based on the STACK EL. In the eighth chapter we provide some experimental results about our implementation of incOSHL.

Chapter 2

Preliminaries

2.1 Classical First-order Logic

In this section, we briefly overview classical first-order logic and its basic concepts. A logic usually consists of formal languages, deduction systems, and semantics. We will look at the formal languages, deduction systems and semantics of classical first-order logic.

2.1 First-order Languages

2.1 Symbols

A first-order language consists of

- Variables
- Function symbols
- Predicate symbols
- Connectives, which usually include $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), and $\supset$(implications)
- Quantifiers, which usually include $\forall$ (for all) and $\exists$ (there exists), and

- Parentheses (and).

Every function symbol has an arity, i.e., the number of argument it takes. A nullary function symbol is sometimes called a constant. Every predicate symbol also has an arity. In classical first-order logic, connectives $\neg$, $\vee$, and $\wedge$, quantifier $\forall$ form a minimum set of connectives and quantifiers from which other connectives and quantifiers can be defined.

2.1 Terms

Terms are defined recursively by the following rules:

1. A variable is a term.
2. If t_1, t_2, …, t_n, where n is a nonnegative integer, are terms, and f is an n-ary function symbol, then $f(t_1, t_2, \ldots, t_n)$ is a term.
3. All terms are formed using Rule 1 or Rule 2.

When $n = 0$, we write f instead of $f()$.

A term is ground if it does not contain any variables. We denote the set of all possible ground terms generated from a finite set F of function symbols by $Terms(F)$. We denote the set of all possible terms generated from a finite set F of function symbols and a finite set V of variable symbols by $Terms(F, V)$.

2.1 Substitutions and Unifiers

A substitution is a function from variables to terms. If a substitution maps only finite number of variables $X_1, \ldots, X_n$ to terms $t_1, \ldots, t_n$ and all other variables to themselves, we write it as $[t_1, \ldots, t_n/X_1, \ldots, X_n]$. We extend a substitution θ to terms as follows:

$$\theta(t) \quad = \quad \begin{cases} t', & t \text{ is a variable and } t' = \theta(t) \\ f(t'_1, \ldots, t'_n) & t = f(t_1, \ldots, t_n) \text{ and } t'_1 = \theta(t_1), \ldots, t'_n = \theta(t_n) \end{cases}$$

We say that a term s is an instance of a term t, if there is a substitution θ such that $s = \theta(t)$.

A substitution σ is a unifier to two terms s and t if $\sigma(s) = \sigma(t)$. It is a most general unifier if for every other unifier σ', there exists a substitution η such that $\sigma'(s) = \eta(\sigma(s))$. For example, the most general unifier of $g(X, f(Y))$ and $g(f(Z), f(f(b)))$ is $[f(Z)/X, f(b)/Y]$, or $[X/Z, f(X)/X, f(b)/Y]$. A substitution is a renaming if it is a one-one mapping from variables to variables. For example, $[Y/X, Z/Y]$ is a renaming, but neither $[Y/X, Y/Z]$ nor $[Y/Z, f(b)/Y]$ is. Usually, there are more then one most general unifiers, but they are all equivalent up to renaming. We denote an arbitrarily chosen most general unifier of two terms s and t by $mgu(s, t)$.

2.1 Atoms

If we consider predicate symbols as a special kind of function symbols that may only occur at the top level of a term, then atoms are terms whose top level function symbol is a predicate symbol.

2.1 Formulas

First-order formulas are defined recursively by the following rules:

1. An atom is a formula.

2. If α is a formula, so is $\neg\alpha$.

3. If both α and β are formulas, so is $\alpha \vee \beta$.

4. If both α and β are formulas, so is $\alpha \wedge \beta$.

5. If both α and β are formulas, so is $\alpha \supset \beta$.

6. If α is a formula and x is a variable, then $\forall x\alpha$ is a formula.

7. If α is a formula and x is a variable, then $\exists x\alpha$ is a formula.

8. All formulas are formed using Rule 1, Rule 2, Rule 3, Rule 4, Rule 5, Rule 6, or Rule 7.

2.1 Deduction Systems

A deduction system usually consists of axioms and rules of inference. In some deduction systems, there are only rules of inference and no axioms. An axiom is usually a formula. Sometimes when there are an infinite number of axioms, a deduction system may provide axiom schemes – templates that can be used to generate the axioms of the deduction system. Rules of inference allow us to derive formulas from formulas. If a formula is an axiom or can be derived from axioms using one or more rules of inference, then the formula is a theorem.

For classical first-order logic, there are several well-known deduction systems, including the Hilbert-style deduction system, natural deduction, and Gentzen's sequent calculus. Most, if not all, commonly used deduction systems for classical first-order logic are equivalent, in the sense that they all derive the same set of theorems.

Automatic theorem provers usually implement none of the foregoing deduction systems. Instead, they use rules of inference that are tailored specifically for mechanized inference.

2.1 Semantics

An interpretation of a first-order language assigns meanings to function symbols and predicate symbols, based on a given domain D. An interpretation assigns to

1. each variable to an object in D
2. each nullary function symbol an object in D,
3. each n-ary function symbol, where $n > 0$, an n-ary function from D^n to D,
4. each nullary predicate symbol a member of $\{TRUE, FALSE\}$, and
5. each n-ary predicate symbol, where $n > 0$, an n-ary function from D^n to $\{TRUE, FALSE\}$.

Given an interpretation of a first-order language, we can determine the truth value of a formula in that first-order language. If an interpretation I makes a formula α true, then we say that I is a model of α, or that I satisfies α, which we denote by $I \models \alpha$; otherwise, we say that I falsifies α, or that α contradicts I, which we denote by $I \not\models \alpha$. If $I \models \alpha$ for every formula α in a set S of formulas, then we write $I \models S$.

A formula or a set of formulas is

- valid if it is satisfied by every interpretation;
- satisfiable is it is satisfied by some interpretation;
- unsatisfiable if it is satisfied by no interpretation.

The dichotomy of syntax and semantics leads to the question of "what is the relation between being provability and validity?" In classical first-order logic and commonly used deduction systems, the answer is simple: a theorem is a valid formula and vice versa. (The proof of this can be found in textbooks) The forward direction is called "soundness" and the converse direction is called "completeness."

2.2 Clauses and Literals

In automatic theorem proving, we use the concepts of literals and clauses.

2.2 Literals

A literal is either an atom or the negation of an atom. A literal is a positive literal if it is an atom. A literal is a negative literal if it is the negation of an atom. The complement $\overline{L}$ of a literal L is defined as $\overline{\neg A} = A$ and $\overline{A} = \neg A$, where A is an atom.

Similar to terms, a literal is ground if it does not contain any variables. We extend a substitution θ to a literal L as follows:

$$\theta(L) = \begin{cases} A', & L \text{ is an atom and } \theta(L) = A' \\ \neg A', & L = \neg A \text{ and } A' = \theta(A) \end{cases}$$

We say that a literal L is an instance of a literal N, if there is a substitution θ such that $L = \theta(N)$.

2.2 Clauses

A clause is a finite set of literals. A formula can be constructed from a non-empty clause in the following way. Given a non-empty clause $\{L_1, \ldots, L_n\}$, with variables $X_1, \ldots, X_m$, the constructed formula is $\forall X_1 \ldots \forall X_m (L_1 \vee \ldots \vee L_n)$. Because in first-order classical logic, universal quantifiers can be switched and $\vee$ is commutative and associative, the order in which the variables and literals appear in the constructed formula does not make any substantial difference, which is consistent with our view of a clause as a set. A clause is ground if all the literals that belong to the clause are ground. We extend a substitution θ to a clause $\{L_1, \ldots, L_n\}$ as follows:

$$\theta(\{L_1, \ldots, L_n\}) \quad = \quad \{L'_1, \ldots, L'_n\} \text{ where } L'_1 = \theta(L_1), \ldots, L'_n = \theta(L_n)$$

We say that a clause C is an instance of a clause D, if there is a substitution θ such that $C = \theta(D)$. We denote the set of all ground instances of all clauses in a set S of clauses by $Gr(S)$.

A ground clause C is true in an interpretation I, written $I \models C$, if and only if $I \models L$ for some $L \in C$. The empty clause is falsified by every interpretation. A set of clauses T is true in I, written $I \models T$, if and only if $I \models D$ for all $D \in T$. Any formula can be converted to the clausal form. When converting a set of formulas to the clausal form, we can preserve their satisfiability, but the resulting set of clauses is not necessarily equivalent to the original set of formulas.

A special class of clauses are Horn clauses. A Horn clause is a clause in which there is at most one positive literal. If a problem does not include non-Horn clauses, then it is usually called a Horn problem; otherwise, a non-Horn problem.

2.3 Automatic Theorem Proving

Automatic theorem proving deals with automating the process of logical inference in a logical system by writing computer programs. In this paper, our logical system will be classical first-order logic.

The completeness of a classical first-order logic says that if a logical formula is valid then there must be a proof for it. In theory, validity in classical first-order logic is semidecidable, which means that there exist algorithms for checking if a formula is valid but there does not exist any algorithm for checking if a formula is invalid. For example, in a Hilbert-style deduction system, it is possible to find a proof for a valid formula by enumerating all possible proofs.

Therefore, theoretically, brute force search algorithms are sufficient, but in practice, they do not work because they are inefficient. Since the search space is extremely large for a such an approach for proof search, it is impossible to prove even some simple problems by brute force search within a reasonable amount of time. The main challenges of automatic theorem proving research are to find proof strategies that can find proofs more efficiently and to find ways to implement these strategies efficiently.

2.3 Herbrand Interpretation

A special class of interpretations is called the Herbrand Interpretations. Given a first-order language, the domain of a Herbrand Interpretation is the set of all terms in that language. A Herbrand Interpretation assigns to

1. each nullary function symbol f f itself,
2. each n-ary function symbol f, where $n > 0$, an n-ary function f^*, such that $f^*(t_1, \dots, t_n) = f(t_1, \dots, t_n)$, i.e., the denotation of a function symbol is a function that constructs new terms using the function symbol itself.
3. each nullary predicate symbol a member of $\{TRUE, FALSE\}$, and
4. each n-ary predicate symbol an set n-ary function from terms to $\{TRUE, FALSE\}$.

The only wildcard here in a Herbrand Interpretation is the interpretation of predicate symbols. Herbrand Theorem says that every satisfiable formula has a

Herbrand Interpretation. This essentially says that we can view Herbrand Interpretations as canonical models for satisfiable formulas. Satisfiability in classical first-order logic is reduced to satisfiability under Herbrand Interpretations.

Objects in the domains of Herbrand Interpretations are purely syntactical and are determined by the input formulas. This property of the Herbrand Interpretation makes it easy for computers to work with interpretations of formulas in a first-order language.

In mathematics, an indicator function is a function defined on a set A to the set $\{TRUE, FALSE\}$. An indicator function indicates membership in a subset of A. If the indicator function maps an element to $TRUE$, then the element is a member of that subset; if the indicator function maps an element to $FALSE$, then the element is not a member of that subset. Given an interpretation of a formal language, the interpretation assigns an element in $\{TRUE, FALSE\}$ to every ground literal in that language. This allows us view an interpretation as an indicator function which indicates membership of the subset of all ground literals to which the interpretation assigns $TRUE$.

In our algorithm, we take a similar approach to representing interpretations. An interpretation I is represented by a set of ground literals. I is a model of a ground literal L if and only if $L \in I$; I is a model of a non-ground literal if and only if it is a model of all ground instances of that literal. We assume in our algorithm that there is an initial interpretation I_0 where satisfiability can be effectively computed. Interpretations can be constructed from I_0 by adding new literals to it. Now, an interpretation I becomes a pair of I_0, an initial interpretation, and a set M of literals that are added to I_0. (I_0, M) is a model of a ground literal L if and only if

1. either $L \in M$,

2. or $\overline{L} \notin M$ and L is satisfied by I_0 (in this case, we can assume that $L \notin M$

since the case $L \in M$ is covered by 1)

The case for non-ground literals is the same as before.

2.3 Proof Strategy

Proof strategies usually also can be expressed in the form of rules of inference. They work in a similar way to deduction systems. However, most proof strategies are not sound and complete in the same sense that a deduction system is sound and complete.

In general, when we say that a proof strategy is sound and complete, we are using the word "sound" and "complete" relative to the input set of formulas and two types of output of the proof strategy. We will denote these two types of output $TRUE$ and $FALSE$:

1. Soundness means that given a set of formulas, if the proof strategy returns $TRUE$, then the set of formulas is satisfiable; if the proof strategy returns $FALSE$ then the set of formulas is unsatisfiable.

2. Given a set of formulas, if it is unsatisfiable, then the proof strategy always returns $FALSE$ in finite number of steps.

2.3 Resolution-based Methods

Resolution [37] is an early proof strategy for classical first-order logic and classical propositional logic that involves essentially one rule of inference – the rule of resolution – and no axioms[1]. To apply the classic resolution [2], all logical formulas

[1]Sometimes for classical first-order logic, factoring is counted as a separate rule

[2]There is a more general version of resolution that does not require this, but it is less commonly used in automatic theorem provers.

have to be converted to the clausal form. The rule of resolution, in its propositional form, looks like

$$\frac{C \cup \{L\} \qquad D \cup \{\overline{L}\}}{C \cup D}$$

where L is a literal, C is a set of literals not containing L, and D is a set of literals not containing $\overline{L}$. The intuition is that the clause $C \cup D$ is a logical consequence of $C \cup \{L\}$ and $D \cup \{\overline{L}\}$ (this can be proved in classical propositional logic).

In resolution, the empty clause indicates $FALSE$ (unsatisfiable); and that no new clauses can be generated indicates $TRUE$ (satisfiable). A derivation from a set of clauses to an empty clause, if it exists, is called a refutation proof. A logical formula is valid if and only if we can find a refutation proof of its negation. For example, $A \supset B$ is valid if and only if we can find a refutation proof of $\neg(A \supset B)$. In this sense, resolution is sound and complete.

The simplicity of the deduction system is where the power of resolution comes from. With resolution, even if we apply brute force search, we are limiting the search to only one possibility in terms of rules of inference and axioms. Of course, we still need to choose on which clauses and how the rule of resolution is to be applied. This is where numerous refinements to resolution came along. Refinements to resolution include the purity principle [38], tautology deletion, subsumption based deletion, unit preference [47], unit resolution, unit resulting resolution, hyper-resolution [39], linear resolution [24, 25], linear resolution with selection function [22], ordered resolution, semantic resolution [42], set of support [48], etc. These refinements help improve the performance of the original resolution to various extents.

Resolution has also been extended with rules to handle equality, including demodulation [49], paramodulation [29, 35], and its restricted version called superposition [6]. E [40] implements the superposition calculus.

One of the problems with resolution-based proof strategies is that they are less efficient on non-Horn problems than on Horn problems [31]. This leads us to the discussion of instance-based methods.

2.3 Instance-based Methods

Instance-based methods are based on instance generation. Resolution may or may not be applied to the generated instances and is not an essential feature. This allows instance-based methods to avoid some of the problems with resolution and provides a different perspective to automatic theorem proving.

An early proof strategy that influenced many instance-based methods is DPLL [13, 12]. The original DPLL algorithm works only for the classical propositional logic, a logic without quantifiers.

There are many variants of DPLL. This brief introduction does not go into the details but summarizes the general ideas behind DPLL. In general, DPLL works on a set S of input clauses and maintains a set M of predicates that represents a partial model such that for any predicate A

- $M \models A$ iff $A \in M$
- $M \not\models A$ iff $\overline{A} \in M$

It starts with M being an empty set, and the goal is to show the satisfiability of S. DPLL usually consists of several rules. No matter which variant it is, the essential idea is basically the same:

- If for some atom A in some clause C in S, neither $M \models A$ or $M \models \overline{A}$, then
  - either add A^d to M
  - or add $\overline{A}^d$ to M

- If for some atom A in some clause C in S, $M \not\models A'$ for all $A' \in C \backslash \{A\}$, then
  - If $M \models A$, do nothing
  - If $M \not\models A$, backtrack to some A'^d, add $\overline{A'}$ to M
  - Otherwise, add A to M.

Intuitively, if some literal is not defined in the model, then it updates the model by arbitrary guessing its truth value. We call this literal a decision literal, which is indicated by the superscript d. If the model makes all but one literal in a clause false, then there are three possibilities for the remaining literal:

- If the model makes the remaining literal true, then it does nothing.
- If the model makes the remaining literal false, then it backtracks to a decision literal and add the complement of that literal to the model as a non-decision literal. If there is no decision literal left, then it returns $FALSE$.
- If the remaining literal is undefined in the model, then it adds that literal to the model.

If the model makes all clauses true, the it returns $TRUE$.

DPLL is sound and complete. An important difference between DPLL and resolution is that while resolution could generate clauses that contain more and more literals, that is not possible for DPLL. This significantly reduces the storage requirement of DPLL. Another difference is that DPLL can be easily integrated with semantics [5], while it seems harder to integrate semantics effectively with resolution. Moreover, DPLL finds a model when it returns $TRUE$, but resolution does not build any model. Modern SAT solvers based on DPLL can handle problems with over one million of variables [17]. Despite the success of DPLL, an obvious drawback of it is that it does not work on classical first-order logic, and there is no easy way to lift it directly to the first-order level, either. However, it inspired a series of development in instance-based proof strategies for classical first-order logic.

Model evolution [9] is a calculus for classical first-order logic which generalizes the ideas of FDPLL [7]. Given a set of input clauses, model evolution maintains a candidate model. In each iteration, it tries to find clauses that are not satisfied by the model and tries to repair the model based on those contradictions. It also has additional simplification rules. One of the key differences between model evolution and OSHL is that the candidate models that model evolution maintain may contain non-ground literals, while the models maintained by OSHL contain only ground literals. Darwin [8] implements the model evolution calculus.

Inst-Gen [21, 15] is a calculus for classical first-order logic that combines resolution with instance generation. Instead of generating the combine clause as the rule of resolution does, Inst-Gen only generates instances of the two clauses being resolved

$$\frac{C \cup \{L\} \qquad D \cup \{\overline{L'}\}}{\theta(C \cup \{L\}) \qquad \theta(D \cup \{\overline{L'}\})}$$

where θ unifies L and L'. The generated clauses are periodically instantiated to ground clauses and a separate proof procedure is used to test the satisfiability of the ground clauses. iProver [20] implements Inst-Gen.

OSHL [33] maintains a model and in each iteration, it tries to find a contradicting ground instance of the input clause that contradicts the current model and updates the current model to satisfy that instance. It allows combining nontrivial semantics with instance generation, which could improve the efficiency of proof search [50]. OSHL differs from other instance-based methods in that it works completely on the ground level. OSHL has been extended with the U rules [27] which provides significant speedup on certain problems [27].

2.4 Order

In this section, we introduce some general notion of order and define orders that are used in this paper.

Definition 1. An order is a pair $\langle D, \leq \rangle$ of a set D and an operator from D^2 to $\{TRUE, FALSE\}$ (We write $a \leq b$ if and only if $\leq (a, b) = TRUE$) such that for every $a, b, c \in D$

1. $a \leq a$
2. $a \leq c$ if $a \leq b$ and $b \leq c$

The order is strict if $a = b$ if $a \leq b$ and $b \leq a$ for all $a, b \in D$.

The order is total if either $a \leq b$ or $b \leq a$ for all $a, b \in D$.

The order is well-founded if there does not exist an infinite sequence $a_1, a_2, \ldots \in D$ such that $a_2 \leq a_1, a_3 \leq a_2, \ldots$.

The order is downward finite if for any element $a \in D$, the set $\{x \leq a \mid x \in D\}$ has finite number of elements.

All orders defined in this section are based on those already defined in the original OSHL paper [33].

2.4 Size Order

First, we define the size order. The size of a term (or an atom) α, written $size(\alpha)$, is defined by the number of occurrences of variables, predicate symbols, and function symbols in that term (or an atom). The size order, written $\leq_s$, on terms or atoms is defined as:

$\alpha \leq_s \beta$ if and only if $size(\alpha) \leq size(\beta)$.

The size order is a well-founded non-strict total order.

2.4 Lexicographic Order

Next, we define the lexicographic order. The lexicographic order can be considered as a dictionary order.

To make the definition clearer, we use the notion of a "flatterm." Given a term (or an atom) α, a flatterm of α, written $flat(\alpha)$ is defined as follows:

1. If α is a variable, then $flat(\alpha) = \alpha$.
2. If α is a nullary function or predicate symbol, then $flat(\alpha) = \alpha$.
3. If α is of the form $x(t_1, \dots, t_n)$, where n is a natural number, and x is an n-ary function or predicate symbol, then $flat(\alpha) = x\ flat(t_1)\ \dots\ flat(t_n)$.

Intuitively, a flatterm is a string of symbols obtained by removing commas and parentheses from a term or an atom. Given a flatterm a, we denote the ith symbol in the flatterm, starting from index 1, by a^i. For example, $flat(f(a,b)) = fab$, and $(fab)^1 = f$, $(fab)^2 = a$, and $(fab)^3 = b$. The size of a term t, written $size(t)$, is the number of symbols in $flat(t)$.

Given a total order $\leq$ on all function symbols and predicate symbols, the lexicographic order [46] with respect to this total order, written $\leq_l$, on terms or atoms is defined as:

$\alpha \leq_l \beta$ if and only if

- α identifies with β or
- there exists an index i such that
  - for all $i' < i$, $flat(\alpha)^{i'} = flat(\beta)^{i'}$
  - and
    * $size(\alpha) = i - 1 < size(\beta)$

- $*$ or $flat(\alpha)^i \leq flat(\beta)^i$

For example, if we use words as examples, we have $star \leq_l starry \leq_l word$.

2.4 Size-lexicographic Order

Now, we can define the size-lexicographic order. The intuition is that it first orders terms (or atoms) by size, then by the lexicographical order. The size-lexicographic order, written $\leq_{sl}$, on terms (or atoms) is defined as: $\alpha \leq_{sl} \beta$ if and only if either $size(\alpha) < size(\beta)$ or $size(\alpha) = size(\beta)$ and $\alpha \leq_l \beta$. On terms (or atoms), $\leq_{sl}$ is a well-founded strict total order.

We can extend this order on atoms to literals as follows: given two literals L and N, we say that $L \leq_{sl} N$ if and only if $A \leq_{sl} A'$, where A is the atom of L and A' is the atom of N, or $N = \neg L$.

2.4 Maximum and Minimum with Respect to An Order

Before we move on to orders on clauses and models, we define several notations that will be useful in the our discussion.

Definition 2. Given an order $\leq$ on a set X, if there is exactly one element e such that $e \leq e'$ for every element e' in some subset Y of X, then we denote this element by $min_{\leq}(Y)$.

Given an order $\leq$ on a set X, if there is exactly one element e such that $e' \leq e$ for every element e' in some subset Y of X, then we denote this element by $max_{\leq}(Y)$.

Given an order $\leq$, we write $\alpha < \beta$ if and only if $\alpha \leq \beta$ but it is not the case that $\beta \leq \alpha$.

We denote the empty set by $\emptyset$. We denote the power set of a set X by $\mathcal{P}(X)$. Given two set X and Y, the set different $X \backslash Y$ denotes the set $\{a \in X \mid a \notin Y\}$.

2.4 Orders on Clauses

The size order can be extended to clauses as follows: Define the size $size(C)$ of a clauses C as the maximum size of any literal in C; given two clauses C and D, $C \leq_s D$ if and only if $size(C) \leq size(D)$.

Since no clause contains both a literal and its negation, the size-lexicographic order can be extended to non-empty clauses as follows: $C \leq_{sl} D$ if and only if $max_{\leq_{sl}}(C) \leq_{sl} max_{\leq_{sl}}(D)$.

2.4 Orders on Models

Now, we define an order on models, which is essential to our proof of completeness of our modified OSHL algorithm. As we defined earlier, a model is a pair of an initial model and a set of ground literals. We define our order on models based on an order on (possibly infinite) sets of ground literals.

Definition 3. The order $\leq_M$ on sets of ground literals is defined as $M_1 \leq_M M_2$ if and only if

1. $M_1 = M_2$ or
2. $M_1 \subset M_2$ or
3. both
   (a) both $M_2 \backslash M_1$ and $M_1 \backslash M_2$ are not empty and
   (b) $min_{\leq_{sl}}(M_2 \backslash M_1) \leq_{sl} min_{\leq_{sl}}(M_1 \backslash M_2)$. (Since $M_2 \backslash M_1$ and $M_1 \backslash M_2$ are disjoint, $min_{\leq_{sl}}(M_2 \backslash M_1) \neq min_{\leq_{sl}}(M_1 \backslash M_2)$))

For example, if we have atoms p, q, r with order $p \leq_{sl} q \leq_{sl} r$ and sets $M_1 = \{p, q\}$ and $M_2 = \{q, r\}$, then we have $min_{\leq_{sl}}(M_2 \backslash M_1) = r$ and $min_{\leq_{sl}}(M_1 \backslash M_2) = p$. Therefore, we have $M_2 \leq_M M_1$.

The ordering $\leq_M$ is extended to models as follows: $(I_0, M_1) \leq_M (I_0, M_2)$ if and only if $M_1 \leq_M M_2$.

2.5 Notations Used in the Type Inference Algorithm

2.5 Vectors

In our discussion of type inference, we will use vectors of sets and functions. In this subsection, we introduce some general notations related to vectors.

We denote the zero-dimensional vector by $\Box$ and an n-dimensional vector with components c_1, ..., c_n by $[c_1, \ldots, c_n]$. We also use a notation for vectors which is similar to the summation operator: $[c_i]_{i=1}^n$ represents vector $[c_1, \ldots, c_n]$. We denote the dimension of a vector $\mathbf{v}$ by $dim(\mathbf{v})$. When we talk about the domain and the codomain of a function, we denote the set of all n-dimensional vectors of objects in a set τ by $[\tau]_n$.

We denote the projection from a vector to its ith component by π_i. We extend projection to sets of vectors as follows:

$$\pi_i(A) \quad = \quad \{\pi_i(\mathbf{v}) \mid \mathbf{v} \in A\}$$

We let all letters and words in bold type run over vectors or functions whose domain is a set of vectors. Following the conventions of mathematics, we identify a one dimensional vector $[a]$ with a scalar a.

Definition 4. To make the presentation concise,

1. Given a domain D, we denote the function $f(x) = \emptyset$ for all $x \in D$ by $\overrightarrow{0}$.

2. We denote the vector $[\overrightarrow{0}]_{i=1}^n$ by $\mathbf{0}_n$. We write $\mathbf{0}$ if n is clear from context.

3. We denote the vector $[\pi_i]_{i=1}^n$ (where the ith component is the projection π_i) by $\mathbf{id}_n$. We write $\mathbf{id}$ if n is clear from context.

4. We denote the vector $[\emptyset]_{i=1}^n$, where all components are the empty set $\emptyset$, by $\mathbf{e}_n$. We write $\mathbf{e}$ if n is clear from context.

If an object a is a component of a vector $\mathbf{v}$, then we write $a \in \mathbf{v}$. Given a vector $\mathbf{v}$ and an object a, denote the vector obtained by replacing the ith component of $\mathbf{v}$ by a by $\mathbf{v}[a/i]$.

We denote the vector formed by appending an object a to the left of an existing vector $\mathbf{v}$ by $a : \mathbf{v}$ and the vector formed by appending an object a to the right of an existing vector $\mathbf{v}$ by $\mathbf{v} : a$. We denote the vector formed joining two vectors $\mathbf{s}$ and $\mathbf{t}$ by $(\mathbf{s}, \mathbf{t})$.

Given a term t, we denote by $\mathbf{v}(t)$ the vector of variables in t, in the order they appear in $flat(t)$. For example, $\mathbf{v}(f(X, Y)) = [X, Y]$.

We define the product function $\prod : [\mathcal{P}(\tau)]_n \rightarrow \mathcal{P}[\tau]_n$ as follows:

$$\prod([A_i]_{i=1}^n) \quad = \quad \{[a_i]_{i=1}^n \mid a_i \in A_i \text{ for } 1 \leq i \leq n\}$$

This operator is similar to the standard Cartesian product, but instead of producing tuples, it produces vectors. For example, $\prod([\{a, b\}, \{c, d\}]) = \{[a, c], [a, d], [b, c], [b, d]\}$.

Given a vector $\mathbf{v}$ of variables and a vector $\mathbf{t}$ of terms with the same dimension, we denote the substitution that substitutes the ith component of $\mathbf{t}$ for the ith component of $\mathbf{v}$, for all possible indices i, by $[\mathbf{t}/\mathbf{v}]$.

2.5 Term Functions and Reverse Term Functions

In this subsection, we assume that there is a fixed set F of function symbols and a fixed set V of variables symbols from which all terms are constructed and abbreviate $Terms(F, V)$ to $Terms$.

Definition 5. Given a term t and a vector of variables $\mathbf{X}$ that contains every variable in $\mathbf{v}(t)$, the term function $\overrightarrow{t_{\mathbf{X}}}$ is a function $[Terms]_{dim(\mathbf{X})} \to Terms$ such that

$$\overrightarrow{t_{\mathbf{X}}}(\mathbf{t}) \quad = \quad t[\mathbf{t}/\mathbf{X}]$$

Intuitively, it substitutes the ith component in $\mathbf{t}$ for the ith variable in $\mathbf{X}$ in term t. For example, $\overrightarrow{f(X)_X}(a) = f(a)$ and $\overrightarrow{f(X)_{[X,Y]}}([a, c]) = f(a)$.

We can extend term functions to $[\mathcal{P}(Terms)]_{dim(\mathbf{X})} \to \mathcal{P}(Terms)$ as follows: Given a set A of vectors of terms,

$$\overrightarrow{t_{\mathbf{X}}}(A) \quad = \quad \{t[\mathbf{t}/\mathbf{X}] \mid \mathbf{t} \in \prod A\}$$

For example, $\overrightarrow{f(X)_X}(\{a, b\}) = \{f(a), f(b)\}$ and $\overrightarrow{f(X)_{[X,Y]}}([\{a, b\}, \{c, d\}]) = \{f(a), f(b)\}$. When $\mathbf{X}$ is clear from context, we may omit $\mathbf{X}$ and write only $\overrightarrow{t}$.

Definition 6. Given a term t and a vector of variables $\mathbf{X}$ that contains every variable in $\mathbf{v}(t)$, the reverse term function $\overleftarrow{t_{\mathbf{X}}}$ is a function $Terms \to \mathcal{P}([Terms]_{dim(\mathbf{X})})$ such that

$$\overleftarrow{t_{\mathbf{X}}}(s) \quad = \quad \{\mathbf{t} \mid s = t[\mathbf{t}/\mathbf{X}]\}$$

For example,

$$\overleftarrow{f(X)_X}(f(a)) \quad = \quad \{a\}$$

and

$$\overleftarrow{f(X)_{[X,Y]}}(f(a)) \quad = \quad \{[a,a],[a,f(a)],[a,f(f(a))],\ldots\}$$

As the second example suggests, a reverse term function $\overleftarrow{t}_{\mathbf{X}}(s)$ may return an infinite set if any variable in $\mathbf{X}$ does not occur in t.

We can extend reverse term functions to $\mathcal{P}(Terms) \rightarrow \mathcal{P}([Terms]_{dim(\mathbf{X})})$ as follows: Given a set B of terms,

$$\overleftarrow{t}_{\mathbf{X}}(B) \quad = \quad \{\mathbf{t} \mid t[\mathbf{t}/\mathbf{X}] \in B\}$$

For example,

$$\overleftarrow{f(X)_X}(\{f(a),f(b),g(c)\}) \quad = \quad \{a,b\}$$

and

$$\overleftarrow{f(X)_{[X,Y]}}(\{f(a),f(b),g(c)\}) \quad = \quad \{[a,a],[b,a],[a,b],[b,b],[a,c],[b,c],\ldots\}$$

When $\mathbf{X}$ is clear from context, we may omit $\mathbf{X}$ and write only $\overleftarrow{t}$.

2.5 Operators on Functions

We denote function composition using operator $\circ$:

$$(f \circ g)(x) \quad = \quad f(g(x)) \tag{2.5.1}$$

and $\underbrace{f \circ \ldots \circ f}_{\text{repeat } n \text{ times}}$ by f^n.

We define a functional union operator $\sqcup$ over functions sharing a common codomain on which the set union operator $\cup$ is defined as follows:

Definition 7. Given two functions f and g of this kind,

$$(f \sqcup g)(x) \quad = \quad f(x) \cup g(x) \tag{2.5.2}$$

It can be easily proved that $\sqcup$ is commutative and associative.

We define a functional subset operator $\sqsubset$ over functions sharing a common codomain on which the subset operator $\subset$ is defined as follows:

Definition 8. Given two such functions f and g with common domain A, $f \sqsubset g$ if and only if for every $a \in A$, $f(a) \subset g(a)$.

It can be easily proved that $\sqsubset$ is transitive.

2.5 Forms of Functions

Definition 9. In our type inference algorithm, we will be dealing with function objects of the following forms:

- Form 1: A term function $\overrightarrow{t}$;
- Form 2: A reverse term function composed with two projections $\pi_p \circ \overleftarrow{t} \circ \pi_q$.

- Form 3: A functional union of finite number of functions of Form 1 or Form 2, i.e., $\bigsqcup_{k=1}^{n} f_k$ where each f_k is a function of the form $\overrightarrow{t}$ or $\pi_p \circ \overleftarrow{t} \circ \pi_q$.

- Form 4: A vector of functions of the form $[f_k]_{k=1}^{n}$ where each f_k is a function of the Form 3.

We define two operators $(-)^{\rightarrow}$ and $(-)^{\leftarrow}$ on the set of all expressions of Form 3. Given an expression F of Form 3, $F^{\rightarrow}$ and $F^{\leftarrow}$ are defined as follows:

$$F^{\rightarrow} = \bigsqcup_{f_k \text{ is of Form 1}} f_k$$
$$F^{\leftarrow} = \bigsqcup_{f_k \text{ is of Form 2}} f_k$$

For example, if $F = \overrightarrow{f(X)_{[X,Y]}} \sqcup \pi_1 \circ \overleftarrow{g(X,Y)_{[X,Y]}} \circ \pi_2$, then $F^{\rightarrow} = \overrightarrow{f(X)_{[X,Y]}}$ and $F^{\leftarrow} = \pi_1 \circ \overleftarrow{g(X,Y)_{[X,Y]}} \circ \pi_2$. By the commutativity of $\sqcup$, $F = F^{\rightarrow} \sqcup F^{\leftarrow}$.

We extend this notation to expressions of Form 4:

$$\mathbf{f}^{\rightarrow} = [(\pi_i(\mathbf{f}))^{\rightarrow}]_{i=1}^{dim(\mathbf{f})}$$
$$\mathbf{f}^{\leftarrow} = [(\pi_i(\mathbf{f}))^{\leftarrow}]_{i=1}^{dim(\mathbf{f})}$$

For example, if $\mathbf{f} = [\overrightarrow{f(X)_{[X,Y]}} \sqcup \pi_1 \circ \overleftarrow{g(X,Y)_{[X,Y]}} \circ \pi_2, \overrightarrow{g'(X,Y)_{[X,Y]}} \sqcup \pi_1 \circ \overleftarrow{f'(X)_{[X,Y]}} \circ \pi_2]$, then $\mathbf{f}^{\rightarrow} = [\overrightarrow{f(X)_{[X,Y]}}, \overrightarrow{g'(X,Y)_{[X,Y]}}]$ and $\mathbf{f}^{\leftarrow} = [\pi_1 \circ \overleftarrow{g(X,Y)_{[X,Y]}} \circ \pi_2, \pi_1 \circ \overleftarrow{f'(X)_{[X,Y]}} \circ \pi_2]$.

2.5 Extension of Operators to Vectors

For every nonnegative integer n, we extend the subset operator $\subset$ to n-dimensional vectors of sets as follows:

$[A_i]_{i=1}^{n} \subset [B_i]_{i=1}^{n}$ if and only if $A_i \subset B_i$ for every $1 \leq i \leq n$.

For example, $[\{a\}, \{c\}] \subset [\{a, b\}, \{c, d\}]$.

We extend the subset operator $\cup$ to n-dimensional vectors of sets as follows:

$$[A_i]_{i=1}^n \cup [B_i]_{i=1}^n \quad = \quad [A_i \cup B_i]_{i=1}^n \tag{2.5.3}$$

For example, $[\{a\}, \{c\}] \cup [\{b\}, \{d\}] = [\{a, b\}, \{c, d\}]$.

We extend the subset operator $\sqcup$ to n-dimensional vectors of functions as follows:

$$[f_i]_{i=1}^n \sqcup [g_i]_{i=1}^n \quad = \quad [f_i \sqcup g_i]_{i=1}^n \tag{2.5.4}$$

For example, $[f, g] \sqcup [f', g'] = [f \sqcup f', g \sqcup g']$.

We extend function application to n-dimensional vectors of functions as follows. Assume that x is in the domain of f_i for $1 \leq i \leq n$.

$$[f_i]_{i=1}^n(x) \quad = \quad [f_i(x)]_{i=1}^n \tag{2.5.5}$$

We extend the function composition operator $\circ$ to n-dimensional vectors of functions as follows. Give a vector $[f_i]_{i=1}^n$ of functions such that for any element x in the domain of $\mathbf{g}$, $\mathbf{g}(x)$ is in the domain of f_i for $1 \leq i \leq n$.

$$[f_i]_{i=1}^n \circ \mathbf{g} \quad = \quad [f_i \circ \mathbf{g}]_{i=1}^n \tag{2.5.6}$$

For example, $[f, f'] \circ [g, g'] = [f \circ [g, g'], f' \circ [g, g']]$. Note that our extension of function composition is asymmetric: it treats the the left operand and the right operand differently. Indeed, $\circ$ is left-distributive over $\sqcup$ but not right-distributive over $\sqcup$.

Lemma 10. $(\mathbf{f} \sqcup \mathbf{g}) \circ \mathbf{h} = \mathbf{f} \circ \mathbf{h} \sqcup \mathbf{g} \circ \mathbf{h}$

Proof. Suppose that

$$\mathbf{f} \quad = \quad [f_1, \ldots, f_n] \tag{2.5.7}$$

and

$$\mathbf{g} \quad = \quad [g_1, \ldots, g_n] \tag{2.5.8}$$

$$\begin{aligned}
(\mathbf{f} \sqcup \mathbf{g}) \circ \mathbf{h}(x) &= [f_i \sqcup g_i]_{i=0}^{n} \circ \mathbf{h}(x) \text{ by } (2.5.4) \\
&= [(f_i \sqcup g_i) \circ \mathbf{h}]_{i=0}^{n}(x) \text{ by } (2.5.6) \\
&= [((f_i \sqcup g_i) \circ \mathbf{h})(x)]_{i=0}^{n} \text{ by } (2.5.5) \\
&= [(f_i \sqcup g_i)(\mathbf{h}(x))]_{i=0}^{n} \text{ by } (2.5.1) \\
&= [f_i(\mathbf{h}(x)) \cup g_i(\mathbf{h}(x))]_{i=0}^{n} \text{ by } (2.5.2) \\
&= [f_i(\mathbf{h}(x))]_{i=0}^{n} \cup [g_i(\mathbf{h}(x))]_{i=0}^{n} \text{ by } (2.5.3) \\
&= [(f_i \circ \mathbf{h})(x)]_{i=0}^{n} \cup [(g_i \circ \mathbf{h})(x)]_{i=0}^{n} \text{ by } (2.5.1) \\
&= [f_i \circ \mathbf{h}]_{i=0}^{n}(x) \cup [g_i \circ \mathbf{h}]_{i=0}^{n}(x) \text{ by } (2.5.5) \\
&= ([f_i \circ \mathbf{h}]_{i=0}^{n} \sqcup [g_i \circ \mathbf{h}]_{i=0}^{n})(x) \text{ by } (2.5.2) \\
&= (\mathbf{f} \circ \mathbf{h} \sqcup \mathbf{g} \circ \mathbf{h})(x) \text{ by } (2.5.6)
\end{aligned}$$

□

Since function application is not right-distributive over $\cup$, for example, $\overrightarrow{f(X,Y)}([\{a\},\emptyset] \cup [\emptyset,\{a\}]) \neq \overrightarrow{f(X,Y)}([\{a\},\emptyset]) \cup \overrightarrow{f(X,Y)}([\emptyset,\{a\}])$, $\circ$ is not right-distributive over $\sqcup$. But we can prove that

Lemma 11. *If $f(A) = \bigcup\{f'(a) \mid a \in A\}$ for some set-valued function f' then $f \circ (\bigsqcup_{k=1}^{\infty} \mathbf{g}_k) = \bigsqcup_{k=1}^{\infty}(f' \circ \mathbf{g}_k)$.*

Proof.

$$\begin{aligned}
(f \circ (\bigsqcup_{k=1}^{\infty} \mathbf{g}_k))(x) &= f((\bigsqcup_{k=1}^{\infty} \mathbf{g}_k)(x)) \\
&= f(\bigcup_{k=1}^{\infty} \mathbf{g}_k(x)) \\
&= \bigcup_{k=1}^{\infty} f'(\mathbf{g}_k(x)) \\
&= \bigcup_{k=1}^{\infty} (f' \circ \mathbf{g}_k)(x) \\
&= \bigsqcup_{k=1}^{\infty} (f' \circ \mathbf{g}_k)(x)
\end{aligned}$$

□

2.5 Path

This subsection introduces the definitions of a path and related terminologies. These terminologies are mainly used in proofs in Section 4.4.3. Essential to the discussion in that subsection is the concept of paths. Using paths allows us to formally express the idea of dealing with different variables in a term separately. Readers can skip this whole subsection if they are not going to read those proofs.

Definition 12. A path is a vector of alternating symbols and integers of the form $[x_0, k_1, x_1, \ldots, k_n, x_n]$, where

1. x_i is a function symbol or predicate symbol for $0 \leq i < n$,

2. x_n is a function symbol, a predicate symbol, or a variable, and

3. k_i is an integer for $0 \leq i \leq n$.

We denote the rightmost component of a path $\mathbf{p}$ by $rm(\mathbf{p})$. Given a term (or an atom), we can generate a unique set of paths from this term (or atom) that contains all information in that term (or atom). Having this set allows us to deal with parts of the terms separately when proving properties about the effect of repeated substitution on terms (or atoms).

Definition 13. The path set of a term (or an atom) t, written $pset(t)$, is defined as follows:

$$\begin{aligned} pset(X) &= \{[X]\} \\ pset(f) &= \{[f]\} \\ pset(f(t_1,\ldots,t_n)) &= \bigcup_{k=1}^{n} \{f : (k : \mathbf{p}) \mid \mathbf{p} \in pset(t_k)\} \end{aligned}$$

The path set of a term (or an atom) contains all paths in that term (or atom). For example, if $t = g(f(a), X)$, then $pset(t) = \{[g, 1, f, 1, a], [g, 2, X]\}$. In order to work with substitutions, we need to define how substitutions are applied to paths.

Definition 14. Given a substitution θ and a path $\mathbf{p}$, the application of θ to $\mathbf{p}$ is defined as

$$\theta(\mathbf{p}) = \begin{cases} \mathbf{p}, & rm(\mathbf{p}) \text{ is not a variable} \\ \{(\mathbf{p}', \mathbf{q}) \mid \mathbf{q} \in pset(\theta(rm(\mathbf{p})))\} & rm(\mathbf{p}) \text{ is a variable}, \mathbf{p} = \mathbf{p}' : rm(\mathbf{p}) \end{cases}$$

For example, if $\theta = [f(a)/X]$, then $\theta([g, 1, f, 1, a]) = \{[g, 1, f, 1, a]\}$ and $\theta([g, 2, X]) = \{[g, 2, f, 1, a]\}$.

Given a substitution θ and a path set P, the result of applying θ on P is defined as:

$$\theta(P) = \bigcup_{\mathbf{p} \in P} \theta(\mathbf{p})$$

To prove properties about the effect of repeated substitution on terms (or atoms), we need the notion of "generators" and "histories" of a path.

Definition 15. We say that a path $\mathbf{p}$ is a generator of a path $\mathbf{p}'$ with respect to θ if $\mathbf{p}' \in \theta(\mathbf{p})$. Given a list $\theta_1, \theta_2, \ldots, \theta_n$ of substitutions, a term (or an atom) t, and a path $\mathbf{p} \in (\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n)(t)$, we define the history of $\mathbf{p}$ with respect to $\theta_1, \theta_2, \ldots, \theta_n$ and t as the list $\mathbf{p}_0, \ldots, \mathbf{p}_n$ such that $\mathbf{p}_n = \mathbf{p}$ and $\mathbf{p}_{k-1}$ is a generator of $\mathbf{p}_k$ with respect to θ_k for $0 < k \leq n$.

For example, if $\theta = [f(a)/X]$, then $[g, 2, X]$ is a generator of $[g, 2, f, 1, a]$ because $\theta([g, 2, X]) = \{[g, 2, f, 1, a]\}$. It is easy to see that if P is a path set of some term (or atom), then the generator of any $\mathbf{p}'$ in P, if it exists, is unique. Finally, we need to notion of "loops" in a substitution.

Definition 16. Given a substitution σ, a loop in σ of length n is a list $\mathbf{p}_0, \ldots, \mathbf{p}_n$ of $n+1$ distinct paths such that $\mathbf{p}_1 \in \sigma(\mathbf{p}_0), \ldots, \mathbf{p}_n \in \sigma(\mathbf{p}_{n-1})$ and $rm(\mathbf{p}_i) \neq rm(\mathbf{p}_j)$ for $0 \leq i < j \leq n$ except $rm(\mathbf{p}_0) = rm(\mathbf{p}_n)$.

In fact, if we have a loop $\mathbf{p}_0, \ldots, \mathbf{p}_n$ with repeated rightmost components between $\mathbf{p}_0$ and $\mathbf{p}_n$, we can always find a smaller loop such that there is no path with repeated rightmost component between them. In this sense, the distinctiveness requirement is not essential in our definition of loops, but it makes some arguments in our proof more concise.

The following lemma shows that substitution is commutative with the *pset* function.

Lemma 17. *For every substitution θ and every term (or atom) t, $\theta(pset(t)) = pset(\theta(t))$.*

Proof. By induction on $size(t)$.

Basis step.

Case 1. t is a variable X. $pset(t) = \{[X]\}$. $\theta(pset(t)) = \{(\Box, \mathbf{q}) \mid \mathbf{q} \in pset(\theta(X))\} = pset(\theta(X))$.

Case 2. t is a constant (or nullary predicate symbol) f. $pset(t) = \{[f]\}$. $\theta(pset(t)) = \{[f]\} = pset(f) = pset(\theta(f))$.

Induction step. Our induction hypothesis (IH) is for all terms (or atoms) with size k or less, the equation we want to prove holds. A term (or an atom) with size $k+1$ must have a function symbol (or a predicate symbol) as its top level symbol. It must have the form $f(t_1, \ldots, t_n)$ for some integer n.

$$\begin{aligned} \theta(pset(t)) &= \theta(\bigcup_{k=1}^{n}\{f:(k:\mathbf{p}) \mid \mathbf{p} \in pset(t_k)\}) \\ &= \bigcup_{k=1}^{n}\theta(\{f:(k:\mathbf{p}) \mid \mathbf{p} \in pset(t_k)\}) \qquad (2.5.9) \end{aligned}$$

Notice that when computing θ on a path, the only thing that may change is the last component of the path and no path can be empty. Therefore, we can extract the common prefix outside the substitution:

$$\begin{aligned} \theta(\{f:(k:\mathbf{p}) \mid \mathbf{p} \in pset(t_k)\}) &= \{f:(k:\mathbf{q}) \mid \mathbf{q} \in \theta(\{\mathbf{p} \mid \mathbf{p} \in pset(t_k)\})\} \\ &= \{f:(k:\mathbf{q}) \mid \mathbf{q} \in \theta(pset(t_k))\} \\ &= \{f:(k:\mathbf{p}) \mid \mathbf{p} \in \theta(pset(t_k))\} \qquad (2.5.10) \end{aligned}$$

$$\begin{aligned}
\theta(pset(t)) &= \bigcup_{k=1}^{n} \theta(\{f:(k:\mathbf{p}) \mid \mathbf{p} \in pset(t_k)\}) \text{ by (2.5.9)} \\
&= \bigcup_{k=1}^{n} \{f:(k:\mathbf{p}) \mid \mathbf{p} \in \theta(pset(t_k))\} \text{ by (2.5.10)} \\
&= \bigcup_{k=1}^{n} \{f:(k:\mathbf{p}) \mid \mathbf{p} \in pset(\theta(t_k)) \text{ by (IH)} \\
&= pset(\theta(t)) \text{ by } \theta(f(t_1,\ldots,t_n)) = f(\theta(t_1),\ldots,\theta(t_n)) \quad (2.5.11)
\end{aligned}$$

We have proved the equation for terms of size $k+1$. $\square$

Chapter 3

genOSHL and incOSHL

While designing incOSHL, I have made several modifications to the original algorithm that are targeted specifically towards obtaining efficient implementations. Some of these modifications are made to reduce repeated computation; some are made to reduce the search space; some are made simply to replace one algorithm-level subroutine that is hard to implement efficiently with another that can be easily implemented efficiently.

In this section, we will first introduce a generalized version of the original OSHL algorithm, genOSHL. genOSHL captures the essential features of OSHL. It has several parameters which can be customized for a specific version of genOSHL. The proof of soundness and completeness of genOSHL can be directly applied for any customization that satisfies certain properties. We also provide details of a customization genOSHL, which we call incOSHL, that is implemented in our theorem prover. We prove the soundness and completeness of incOSHL using the framework we set up in genOSHL. We also compare incOSHL with the original OSHL algorithm and take a look at the changes that are incorporated into incOSHL.

3.1 OSHL

In this section, we briefly review the original OSHL algorithm [33]. The basic idea of OSHL is as follows: Given an initial interpretation I_0, a set S of input

T	M	D
$\emptyset$	$\emptyset$	$\{p,q\}$
$\{p,q\}$	$\{q\}$	$\{p,\neg q\} \Rightarrow \{p\}$
$\{p\}$	$\{p\}$	$\{\neg p,q\} \Rightarrow \{q\}$
$\{p\} \quad \{q\}$	$\{p,q\}$	$\{\neg p,\neg q\} \Rightarrow \emptyset$

Table 3.1: Example 1 of the Original OSHL Algorithm

clauses, and a size-lexicographical order $\leq_{sl}$ on ground literals, it maintains a set M of literals and a set T of ground instances of clauses in S. In each iteration, it finds a minimal instance D with respect to $\leq_{sl}$ that contradicts (I_0, M) and updates M and T, until either such a D cannot be found or the empty clause can be derived from T.

When OSHL updates M and T with D, it first performs ordered resolution using D as the main premise and clauses in T as side premises. Ordered resolution on ground clauses is defined as follows: if ground clause C has maximum literal L and ground clause D has maximum literal $\overline{L}$, both with respect to $\leq_{sl}$, then the ordered-resolvent of C and D, denoted by $AR(C, D)$, is $(C\backslash\{L\}) \cup (D\backslash\{\overline{L})\}$. After performing ordered resolution, it inserts the resolvent D' to T and deletes all clauses in T that are greater than D' with respect to $\leq_{sl}$. For example, if T =$\{\{p(a), q(a, c)\}, \{q(a, b), p(f(f(a)))\}\}$ and D =$\{\neg p(f(f(a)))\}$, then D' =$\{q(a, b)\}$ and after adding D' to T and deleting clauses that are greater than D', we have T =$\{\{p(a), q(a, c)\}, \{q(a, b)\}\}$.

M is generated by taking all the maximum literals of clauses in T. For example, if T =$\{\{p(a), q(a)\}, \{q(a), p(f(a))\}\}$, then M =$\{q(a), p(f(a))\}$.

The following is an example run of OSHL: Suppose that $p \leq_{sl} q$, I_0 makes all negative literals true, and $S = \{\{p, q\}, \{\neg p, \neg q\}, \{p, \neg q\}, \{\neg p, q\}\}$. The steps run by the prover are shown in Table 3.1. The first non-header row shows the initial value of T and M and the first contradicting instance D. The second row shows

T	M	D
$\emptyset$	$\emptyset$	$\{p, q\}$
$\{p, q\}$	$\{q\}$	$\{p, \neg q\} \Rightarrow \{p\}$
$\{p\}$	$\{p\}$	$\{\neg p, q\} \Rightarrow \{q\}$
$\{p\} \quad \{q\}$	$\{p, q\}$	

Table 3.2: Example 2 of the Original OSHL Algorithm

T	M	D
$\emptyset$	$\emptyset$	$\{p, q\}$
$\{p, q\}$	$\{q\}$	$\{r\}$
$\{p, q\} \quad \{r\}$	$\{q, r\}$	$\{p, \neg q, \neg r\} \Rightarrow \{p\}$
$\{p\}$	$\{p\}$	$\{r\}$ (repeated)
$\{p\} \quad \{r\}$	$\{p, r\}$	$\{\neg p, \neg r\} \Rightarrow \emptyset$

Table 3.3: Example 3 of the Original OSHL Algorithm

the updated T and M by adding D on the first row to T and generating M from T. The contradicting instance D on the second row can be resolved with the clause in T on the second row, resulting in a new clause D'. This is denoted by $D \Rightarrow D'$. After adding D' to T, the original clause in T needs to be deleted, the resulting T is shown on the third row. Repeating this process, on the fourth row the empty clause is generated which shows that S is unsatisfiable.

Table 3.2 shows an example of running OSHL on a satisfiable problem. This problem is the same as the previous problem except that we deleted a clause from S, making it satisfiable: Suppose that $S = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}\}$. The steps are similar to those of the previous problem, but instead of generating the empty clause, they generate a model for S, which shows that S is satisfiable.

Table 3.3 shows a motivating example of why we want to tweak the original OSHL algorithm. This table shows the steps for proving the following problem: $p \leq_{sl} q \leq_{sl} r$, I_0 makes all negative literals true, and $S =\{\{p, q\}, \{r\}, \{p, \neg q, \neg r\}, \{\neg p, \neg r\}\}$. In this example, the instance $\{r\}$ is generated twice, the reason being that after it was first generated and added to T, another clause $\{p\}$ is generated and added to T and forced $\{r\}$, which is lexicographically greater than $\{p\}$, to be deleted from T. This is redundant computation since $\{r\}$ does not have any link with $\{p\}$. In our modified version of our algorithm, we will not require T to be ordered. Instead, we maintain a more relaxed, but more complex property called "perfect linking."

3.2 genOSHL

The motivation of creating genOSHL is as follows: The original OSHL is sound and complete, but in order to find an efficient implementation, some subroutines in the original OSHL algorithm need to be tweaked. We would like to find a systematic way of figuring out how the tweaking affects soundness and completeness of the resulting algorithm. In this section, we introduce the genOSHL framework, of which both the original OSHL algorithm and our variant, incOSHL, are instances.

The genOSHL algorithm works on a fixed set of input clauses and generates a sequence of instances and models. In the following discussion, we will always assume the following implicit notations: the fixed set of input clauses is S and the initial interpretation is I_0.

The main proof procedure builds a sequence of models. The newest model is called the current model. The algorithm alternates between two modes. In the first mode, it tries to generate an instance that is contradictory to the current model; in the second mode, it tries to generate a new model by adjusting the current model so that the new model satisfies the generated instance. If the prover fails to generate a contradicting instance in the first mode, then it has found a model for the input clauses; if it fails to generate a new model that satisfies the generated instance, then it has found a refutation proof for the input clauses.

Now, we formalize this algorithm. We will describe a general OSHL algorithm where we intentionally leave some functions (d, $simp$, m) undefined. Instead, we will list properties that these functions should have. This gives us flexibility to design the details of those functions, while guaranteeing that whatever functions we design, as long as they have those properties, the customized genOSHL algorithm is still sound and complete.

Definition 18. Given a set S of input clauses, an initial interpretation I_0, genOSHL constructs a sequence of ground clauses D_i, a sequence of sets T_i of ground instances of the input clauses, and a sequence of sets M_i of ground literals, where $i \in \mathbb{N}$, as follows:

$$D_0 = \emptyset \tag{3.2.1}$$

$$T_0 = \emptyset \tag{3.2.2}$$

$$M_0 = \emptyset \tag{3.2.3}$$

$$D_{k+1} = d(S, I_0, M_k) \text{ s.t. } (I_0, M_k) \not\models D_{k+1} \tag{3.2.4}$$

$$T_{k+1} = simp(T_k, D_{k+1}) \text{ s.t. } T_k \cup \{D_{k+1}\} \models T_{k+1} \tag{3.2.5}$$

$$M_{k+1} = m(T_{k+1}) \text{ s.t. } M_{k+1} \models T_{k+1} \tag{3.2.6}$$

Intuitively, genOSHL starts with an empty set and an initial model. In each iteration, the prover tries to find a ground instance that contradicts the current model, then it adjusts the current model to satisfy the contradicting ground instance. In this construction, d is a partial function that chooses a ground instance of one of the input clauses such that it contradicts the current model. $simp$ is a function that takes in a set of ground clauses together with another ground clause, and returns a new set of ground clauses which are logical consequences of the input of this function. m is a partial function that takes in a set of clauses and returns a model of it. These three functions are left undefined, so that we can customize genOSHL later.

The generation of these sequences is controlled by the following recursive function:

$$genOSHL(T_k; S) = \begin{cases} FALSE, & T_k \vDash \bot \\ TRUE, & M_k \vDash S \\ genOSHL(T_{k+1}; S), & \text{otherwise} \end{cases}$$

where we denote "false" by $\bot$. $genOSHL$ is a partial function. There are three possible outcomes of $genOSHL$.

1. If D_{k+1} in (3.2.4) cannot be found, then it means that we cannot find any ground instance that contradicts the current model, i.e., the current model is a model of the input clauses, i.e., $M_k \models S$. Hence the input clauses are satisfiable.

2. If M_{k+1} in (3.2.6) cannot be constructed, then it means that we cannot adjust the current model to satisfy the contradicting clauses, i.e., the set T_k in unsatisfiable. Hence the input clauses are unsatisfiable;

3. It runs forever.

We will show later that the algorithm is complete, i.e., if the input clauses are unsatisfiable, the third case will never occur.

We will elaborate on some additional properties that the three functions in Definition 18 should have in order to guarantee soundness and completeness of genOSHL.

3.2 Function d

The function d has three parameters: S the set of input clauses, I_0 the initial interpretation, and M a set of ground literals. d returns an instance of some clause in S which is contradictory to (I_0, M). There are many ways to fill in the details of d, but to ensure the completeness of the algorithm, d should satisfy the following minimality requirement:

Proposition 19. *If S is unsatisfiable, then there is*

1. *an unsatisfiable subset S' of $Gr(S)$ and*

2. *a downward finite order $\leq_{df}$ on S'*

such that $d(S, I_0, M)$ always chooses a ground instance in S' that is minimal with respect to $\leq_{df}$.

When specifying d, we may also specify the set S' and $\leq_{df}$, as we will see in our discussion of type inference. $\leq_{df}$ only needs to be downward finite on S' and needs not to be downward finite on any of its extensions. In our untyped algorithm, S' is $Gr(S)$; in our typed algorithm, S' is usually a proper subset of $Gr(S)$. It can be easily seen that if $\leq_{df}$ is downward finite on $Gr(S)$, then it is downward finite on its restrictions on subsets of $Gr(S)$.

One candidate of $\leq_{df}$ is $\leq_{sl}$. Another candidate of $\leq_{df}$ is $\leq_s$. If we use $\leq_s$ as the $\leq_{df}$, then there can be a multiple but finite number of ground instances that satisfy the minimality requirement. We may make d arbitrarily choose any one of those ground instances.

3.2 Function *simp* and Function m

Similar to d, there are many ways to fill in the details of *simp* and m, but to ensure the completeness of the algorithm, we cannot arbitrarily choose *simp* and m in our construction, either. We choose *simp* and m so that

Proposition 20. *simp, m, and M_k have the following properties:*

$$M_k \quad <_S \quad M_{k+1} \tag{3.2.7}$$

$$\bigcup simp(T, D) \quad \subset \quad \bigcup(T \cup \{D\}) \tag{3.2.8}$$

$$max_{\leq_{sl}}(m(T)) \quad \leq_{sl} \quad max_{\leq_{sl}}(T) \tag{3.2.9}$$

Here, we use the notation $\bigcup T$ for $\bigcup_{x \in T} x$.

Intuitively, the first property says that the sequence of models grows strictly monotonically. This is crucial in our proof of completeness as it ensures that if S is unsatisfiable, then the condition in (3.2.6) will eventually be falsified. This monotonicity is a key property in the original OSHL algorithm, which is used in its completeness proof [50]. We extended it to genOSHL. The second property says that the function *simp* should not introduce new literals that do not already appear in its arguments; the third property says that the literals in a model generated by m should not be larger than the largest literal in its argument, with respect to $\leq_{sl}$, so that we have a finite search space for the models.

3.3 Soundness and Completeness of genOSHL

3.3 Soundness

Now, we prove the soundness of genOSHL.

Theorem 21. *If genOSHL halts, returning unsatisfiable, then S is unsatisfiable; if genOSHL halts, returning satisfiable, then S is satisfiable.*

Proof. Case 1. If the genOSHL procedure returns unsatisfiable at the ith recursion, then we know that T_i is an unsatisfiable set of clauses. We construct a new sequence of sets of ground clauses T'_k such that all clauses in T'_k are ground instances of clauses in S and $T'_k \models T_k$ for $k = 1, \ldots, i$.

$$
\begin{aligned}
T'_0 &= \emptyset & (3.3.1)\\
T'_{k+1} &= T'_k \cup \{D_{k+1}\} & (3.3.2)
\end{aligned}
$$

where D_{k+1} was defined in (3.2.4). It can be easily seen that T'_k is a set of ground instances of clauses in S, since all D_ks are. Now we prove by induction that $T'_k \models T_k$.

Basis step. $T_0 = T'_0 = \emptyset$. Therefore, it is trivially true that $T_0 \models T'_0$.

Induction step. Our Induction Hypothesis (IH) is that $T'_k \models T_k$, by the condition in (3.2.5), $T_k \cup \{D_{k+1}\} \models T_{k+1}$. By IH, and properties of classical first-order logic, $T'_k \cup \{D_{k+1}\} \models T_{k+1}$. By definition, $T'_{k+1} \models T_k$.

We have proved that $T'_k \models T_k$ for $k = 1, \ldots, i$. Now, since T_i is unsatisfiable, T'_i must also be unsatisfiable. Since T'_i contains only instances of clauses in S, S is also unsatisfiable.

Case 2. If the genOSHL procedure returns satisfiable at the ith recursion, then we know that M_i is a model of S, which is just another way of saying that S is a satisfiable. □

3.3 Completeness

We need to show that

Theorem 22. *if S is unsatisfiable, then genOSHL always halts.*

Proof. We prove this by contradiction. We use the following convention: Given a set A of sets, we denote the union of all sets in A by $\bigcup A$.

Suppose towards a contradiction that S is unsatisfiable and genOSHL never halts on input S. We denote the subset and the downward finite order in Proposition 19 by S' and $\leq_{df}$ respectively.

By compactness of classical first-order logic, since S' is unsatisfiable, we can find a finite unsatisfiable set T of ground instances of clauses in S'. This set T is not satisfied by any model, including the infinite series of models M_k constructed by genOSHL. This means that for every model M_k there must be some clause D^* in T such that M_k falsifies D^*. By the minimality requirement of d, for every ground instance D chosen by d, $D \leq_{df} D^*$ for some D^* in T. By the downward-finiteness of $\leq_{df}$ and finiteness of T, there are only finite number of ground instances from which d can choose D. We denote this finite set of ground instances by T^*. Since each instance in T^* is also a clause, which is a finite set of literals, the set of literals $\bigcup T^*$ is also finite.

Next, we can prove by induction that $\bigcup T_k \subset \bigcup T^*$ for all $k \in \mathbb{N}$.

Basis step. $T_0 = \emptyset$. It is trivially true that $\bigcup T_0 \subset \bigcup T^*$.

Induction step. Our Induction Hypothesis (IH) is that $\bigcup T_k \subset \bigcup T^*$. By construction (3.2.5), $T_{k+1} = simp(T_k, D_{k+1})$. By definition, $D_{k+1} \in T^*$, hence $D_{k+1} \subset \bigcup T^*$. By (IH), $\bigcup T_k \subset \bigcup T^*$. Hence, $\bigcup(T_k \cup \{D_{k+1}\}) \subset \bigcup T^*$. By (3.2.8), $\bigcup T_{k+1} \subset \bigcup(T_k \cup \{D_{k+1}\})$. By transitivity of $\subset$, $\bigcup T_{k+1} \subset \bigcup T^*$.

We have proved that $\bigcup T_k \subset \bigcup T^*$ for all $k \in \mathbb{N}$.

By (3.2.9), we have

$$max_{\leq_{sl}}(M_k) \quad \leq_{sl} \quad max_{\leq_{sl}}(\bigcup T^*) \tag{3.3.3}$$

for all $k \in \mathbb{N}$.

On the other hand, the series M_k of models is monotonically increasing with respect to strict order $<_M$, i.e., there are infinitely many models in this series. But given a finite set of literals, we can only construct a finite set of models. By its contrapositive, $\bigcup_{k=0}^{\infty} M_k$ must contains infinitely many literals. By downward-finiteness of $\leq_{sl}$, we have a contradiction since (3.3.3) would require $\bigcup_{k=0}^{\infty} M_k$ to contain only finitely many literals. $\square$

3.4 incOSHL

Having defined genOSHL and proved its soundness and completeness, we now customize genOSHL into incOSHL by filling in the details of the functions d, $simp$, and m.

3.4 Function d

The function d used in the incOSHL algorithm finds a minimal ground instance of some clause in S with regard to $\leq_s$ that contradicts the current model, i.e. $d(S, I_0, M) = D$ such that for every ground instance D' of some clause in S such that $(I_0, M) \not\models D'$, $D \leq_s D'$. Usually, there are multiple ground instances that satisfy this minimality condition. We may arbitrarily choose any one of those ground instances. In practice, d runs a search subroutine and chooses a first instance found. Next, we show that

Lemma 23. *$\leq_s$ is downward finite on $gr(S)$.*

Proof. We only need to show that given a natural number N, there are finitely many ground instances of S with size less than or equal to N. Since S is a finite set of clauses, the goal can be reduced to showing that given a natural number N and a clause C, there are finitely many ground instances of C with size less than or equal to N. Suppose we have a grounding substitution σ. Recall that the size of $\sigma(C)$ is

$$size(\sigma(C)) \quad = \quad max_{\leq}(\{size(L) \mid L \in \sigma(C)\})$$

where $\leq$ is the usual order on integers. Since $\sigma(C)$ is always a finite set of ground literals, the goal can be further reduced to showing that given a natural number N and a literal L, there are finitely many ground instances of L with size less than or equal to N.

Notice that the function *flat* defined in Section 2.4 is an inclusion, which means that the cardinality of a set of ground literals with size less than or equal to N cannot be larger than that of a set of string of predicate symbols and function symbols with length less than or equal to N, which is finite, given a finite number of predicate symbols and a finite number of function symbols. $\square$

To illustrate the function d, let us take a look at an example.

Suppose that $S = \{\{\neg p(X)\}, \{\neg p(f(Y))\}, \{p(a)\}, \{p(f(a))\}\}$, I_0 makes all negative literals true, and the current model is $(I_0, \{p(a), p(f(a))\})$. d returns a minimal instance with respect $\leq_s$ that contradicts the current model. We can generate the contradicting instances in three ways:

1. instantiating X to a in $\{\neg p(X)\}$,

2. instantiating X to $f(a)$ in $\{\neg p(X)\}$, and

3. instantiating Y to a in $\{\neg p(f(Y))\}$.

1 will produce $\{\neg p(a)\}$, both 2 and 3 will produce $\{\neg p(f(a))\}$. By the minimality requirement, d returns $\{\neg p(a)\}$.

3.4 Function *simp* and Function m

3.4 The Construction

Now, we describe the function m and *simp* used in the incOSHL algorithm. The function m maps a set T of ground clauses to the set of all maximum literals of clauses in T. More formally,

Definition 24. The m function of *incOSHL* is defined as

$$m(T) \quad = \quad \{max_{\leq_{sl}}(D) \mid D \in T\}$$

Given an arbitrary set T, it is not always the case that $(I_0, m(T)) \models T$. For example, if we had $p \leq_{sl} q$, $T = \{\{q\}, \{p, \neg q\}\}$, and I_0 made all negative literals true, $m(T)$ would be $\{\neg q\}$, which when combined with I_0 would not produce a model for T. In order for $m(T)$ to be a model of T, both $\overline{L}$ and L cannot be the maximum literals of some clauses in T. This example shows the need for a *simp* function that avoids this situation. In particular, we can achieve this by ensuring that the maximum literal of every clause in T is false in I_0.

Lemma 25. *If no clause in T contains both a literal and its complement, and $I_0 \not\models max_{\leq_{sl}}(D)$ for every clauses D in T, then $(I_0, m(T)) \models T$.*

Proof. By definition of models, for every clause D in T, $max_{\leq_{sl}}(D) \in m(T)$. Therefore, $(I_0, m(T)) \models D$. □

The function application $simp(T_{k-1}, D_k)$ works in two steps. In Step 1, *simp* performs ordered resolution with respect to $\leq_{sl}$.

simp repeatedly performs ordered resolution with respect to $\leq_{sl}$ on D_k and clauses in T_{k-1}. This step can be formally specified by the construction of a series $D'_{k,s}$ of ground clause as follows

$$D'_{k,0} \quad = \quad D_k \tag{3.4.1}$$

$$D'_{k,s+1} \quad = \quad AR(D'_{k,s}, E) \text{ where } E \in T_{k-1} \tag{3.4.2}$$

If an empty clause is reached, then the incOSHL stops under halting condition that T_{k-1} is unsatisfiable. If a nonempty clause is reached, it proceeds to Step 2. In Step

2, we denote this nonempty clause by D'_k.

In Step 2, *simp* calls a subalgorithm called *delete*. *delete* constructs T_k by deleting some ground clauses from T_{k-1} and adding D'_k. To do so,

Definition 26. Given T_{k-1} from Definition (18), we constructs a sequence $E_{k,s}$ of ground clauses, a sequence $A_{k,s}$ of sets of ground literals, and a sequence $U_{k,s}$ of sets of ground clauses as follows:

$$\begin{aligned} E_{k,0} &= \emptyset && (3.4.3)\\ A_{k,0} &= \{max_{\leq_{sl}}(D'_k)\} && (3.4.4)\\ U_{k,0} &= T_{k-1} && (3.4.5)\\ E_{k,s+1} &= E \text{ where } E \in U_{k,s} \text{ and } A_{k,s} \cap E \neq \emptyset && (3.4.6)\\ A_{k,s+1} &= A_{k,s} \cup \{\overline{max_{\leq_{sl}}(E_{k,s+1})}\} && (3.4.7)\\ U_{k,s+1} &= U_{k,s} \backslash \{E_{k,s+1}\} && (3.4.8) \end{aligned}$$

Intuitively, the construction recursively removes clauses from $U_{k,s}$ that contain some literal in $A_{k,s}$ and adds the complement of the maximum literals of the removed clause to the sets $A_{k,s+1}$. The effect of this construction is that it removes any clause $E_{k,s+1}$ that is redundant in the following sense: a model of the set $U_{k,s} \cup \{D'_k\} \backslash \{E_{k,s+1}\}$ generated by applying m to this set is also a model of $U_{k,s+1} \cup \{D'_k\}$.

We give the definition of *delete* as a recursive function:

Definition 27. The *delete* function of incOSHL is defined as

$$delete(U_{k,s}, A_{k,s}; D'_k) = \begin{cases} delete(U_{k,s+1}, A_{k,s+1}; D'_k), & E_{k,s+1} \text{ exists} \\ U_{k,s} \cup \{D'_k\}, & \text{otherwise} \end{cases}$$

We give the definition of *simp* as a recursive function:

i	$E_{k,i}$	$A_{k,i}$	$U_{k,i}$
0	$\emptyset$	$\{p\}$	T_{k-1}
1	$\{p,q\}$	$\{p,\neg q\}$	$\{\{p,r\},\{\neg r,r'\}\}$
2	$\{p,r\}$	$\{p,\neg q,\neg r\}$	$\{\{\neg r,r'\}\}$
3	$\{\neg r,r'\}$	$\{p,\neg q,\neg r,\neg r'\}$	$\emptyset$

Table 3.4: An Example: *delete*

Definition 28. The *simp* function of incOSHL is defined as

$$simp(T_{k-1}, D'_{k,s}) = \begin{cases} simp(T_{k-1}, D'_{k,s+1}), & D'_{k,s+1} \text{ exists} \\ delete(T_{k-1}, \{max_{\leq_{sl}}(D'_{k,s})\}; D'_{k,s}), & \text{otherwise} \end{cases}$$

To illustrate the functions, let us take a look at an example. Suppose that

1. I_0 makes all negative literals true,
2. we are constructing T_k,
3. we have atoms $p \leq_{sl} q \leq_{sl} r \leq_{sl} r'$, and
4. $T_{k-1} = \{\{p,q\},\{p,r\},\{\neg r,r'\}\}$.

The current model generated by m is $(I_0, \{q,r,r'\})$. Suppose that d returns an instance $\{\neg q\}$. In the first step of *simp*, $\{\neg q\}$ resolves with $\{p,q\}$. The resolvent $\{p\}$ cannot be resolved with any clauses in T_{k-1} any more. *simp* goes to the second step with $D'_k = \{p\}$. As illustrated in Table 3.4, $A_{k,0} = \{p\}, U_{k,0} = T_{k-1}$. Since $A_{k,0} \cap \{p,q\} \neq \emptyset$, we delete $\{p,q\}$ and set $A_{k,1} = \{p,\neg q\}, U_{k,1} = \{\{p,r\},\{\neg r,r'\}\}$. Since $A_{k,1} \cap \{p,r\} \neq \emptyset$, we delete $\{p,r\}$ and set $A_{k,2} = \{p,\neg q,\neg r\}, U_{k,2} = \{\{\neg r,r'\}\}$. Finally, since $A_{k,2} \cap \{\neg r,r'\} \neq \emptyset$, we delete $\{\neg r,r'\}$ and set $A_{k,3} = \{p,\neg q,\neg r,\neg r'\}, U_{k,3} = \emptyset$, and we have $T_k = U_{k,3} \cup \{D'_k\} = \{\{p\}\}$.

3.4 Perfect Linking

Next, let us consider the general notion of "perfect linking." incOSHL maintains the "perfect linking" property of T_k in Definition (18) through ordered resolution and *delete*. This allows us to retain more clauses in T_k than the original OSHL algorithm.

Definition 29. A ground clause D has the "perfect linking" property (or is perfectly linked for short) with respect to a set U of ground clauses, if for every literal L in D,

1. if $I_0 \models L$, then there exists exactly one clause $E \in U$ such that $L = \overline{max_{\leq_{sl}}(E)}$;

2. if $I_0 \not\models L$, there there exists no clause in $U \backslash \{D\}$ such that $L = max_{\leq_{sl}}(E)$.

The idea of perfect linking is the following: Suppose the initial interpretation makes all negative literals true. There is a set of literals M consists of the maximum literals of all clauses in U. a clause D is perfectly linked if in D

1. no positive literals are in M, and

2. every negative literals has its complement in M.

In other words, if the current model is (I_0, M), then the D is contradictory to the current model.

For a set to be perfectly linked, every clause in it has to be perfectly linked with respect to the rest of the set.

Definition 30. A set U of ground clauses has the "perfect linking" property, if for every clause E in U

1. E has the "perfect linking" property with respect to the set itself, and

2. $I_0 \not\models max_{\leq_{sl}}(E)$.

For example, suppose that we have

1. I_0 makes all negative literals true,

2. atoms $p \leq_{sl} q \leq_{sl} r \leq_{sl} r'$, and

3. $U = \{\{p, q\}, \{p, r\}, \{\neg r, r'\}\}$.

Then, both $\{p\}$ and $\{p, \neg q\}$ are perfectly linked with regard to U, but neither $\{\neg p, \neg q\}$ nor $\{p, q\}$ is since they falsify the first and the second condition, respectively. The set U itself is perfectly linked. For example, in $\{\neg r, r'\}$, r is the maximum literal in $\{p, r\}$, and r' is the maximum literal of no other clause then itself and r' is false in I_0. However, if we had $U = \{\{p, \neg q\}, \{p, r\}, \{\neg r, r'\}, \{p\}\}$, it would not be perfectly linked since neither $\{p, \neg q\}$ nor $\{p, r\}$ would be perfectly linked with respect to U.

The main purpose of introducing the perfect linking property is to reduce the number of repeated computations in incOSHL. In the original OSHL algorithm, if we generate the following clauses in order: $\{p, q\}$, $\{r\}$, $\{\neg r, r'\}$, $\{p\}$, then after $\{p\}$ is generated, all other three clauses are deleted because they are all lexicographically larger. incOSHL only requires all the generated clauses except those that are deleted to form a perfectly linked set. The only clause that needs to be deleted is $\{p, q\}$. This allows us to retain $\{r\}$ and $\{\neg r, r'\}$, thereby avoiding generating them again.

Next, we prove some properties of perfectly linked ground clauses and perfectly linked sets.

Lemma 31. *If a ground clause D is perfectly linked with respect to U, then D cannot contain both a literal and its complement.*

Proof. Suppose towards a contradiction that there is a literal L such that $L, \overline{L} \in D$. Without loss of generality, assume that $I_0 \models L$. On the one hand, by Clause 1 of Definition 29, there is a clause E in U such that $L = \overline{max_{\leq_{sl}}(E)}$. On the other hand, by Clause 2 of Definition 29, there is no clause E in U such that $\overline{L} = max_{\leq_{sl}}(E)$. This is a contradiction. □

Next, we show that ordered resolution preserves the "perfect linking" property.

Lemma 32. *If ground clauses C and D are both perfectly linked with respect to U and they are ordered-resolvable, then $AR(C, D)$ is also perfectly linked with respect to U.*

Proof. For every literal L in $AR(C, D)$, L belongs to at least one of C and D. Without loss of generality, we assume that L belongs to C.

1. If $I_0 \models L$, then by the premise that C is perfectly linked, there exists exactly one clause $E \in U$ such that $L = \overline{max_{\leq_{sl}}(E)}$.

2. If $I_0 \not\models L$, then by the premise that C is perfectly linked, there there exists no clause in $U \backslash \{C\}$ such that $L = max_{\leq_{sl}}(E)$. We need to show that there exists no clause in $U \backslash \{AR(C, D)\}$ such that $L = max_{\leq_{sl}}(E)$. To show this, we only need to show that there exists no clause in $U \backslash \{AR(C, D)\} \backslash (U \backslash \{C\})$ such that $L = max_{\leq_{sl}}(E)$. Since $U \backslash \{AR(C, D)\} \backslash (U \backslash \{C\}) \subset \{C\}$, we only need to show that $L \neq max_{\leq_{sl}}(C)$. L cannot be the maximum literal in C because $L \in AR(C, D) \cap C = C \backslash max_{\leq_{sl}}(C)$.

□

Corollary 33. *If*

1. *T is a perfectly linked set of ground clauses,*

2. D is a perfectly linked ground clause with respect to T, and

3. D' is obtained from performing zero or more steps of ordered resolution using D as the main premise and clauses in T as side premises,

then D' is also perfectly linked with respect to T.

Proof. By induction on the number of ordered resolutions performed. □

3.4 Properties of m in Definition 24

Now consider a set $T \cup \{D\}$, even if T is perfectly linked and D is perfectly linked with respect to T, this set may not be perfectly linked. For example, if I_0 make all negative literals true, $p \leq_{sl} q \leq_{sl} r \leq_{sl} r'$, $T = \{\{q, r\}, \{\neg r, r'\}\}$, and $D = \{p, q\}$, then $T \cup \{D\}$ is not perfectly linked even though T is perfect linked and D is perfectly linked with respect to T. The problem here is that adding D to the set T causes q in clause $\{q, r\}$ to violate Clause 2 in Definition 29. Even if we delete the clause $\{q, r\}$ from T, the resulting set is still not perfectly linked, as $\neg r$ in clause $\{\neg r, r'\}$ now violates Clause 1 in Definition 29. We need to delete this clause, too, if we want to restore the "perfect linking" property. In fact, it is sufficient to perform these deletions repeatedly until no violation exists, as captured in *delete*. We will prove that this is indeed the case.

First, we show that the initial model I_0 must falsify the maximum literal of a perfectly linked clause that cannot be further ordered-resolved.

Lemma 34. *If D is a perfectly linked ground clause with respect to T and not ordered resolvable with any ground clause in T, then $I_0 \not\models max_{\leq_{sl}}(D)$.*

Proof. Suppose towards a contradiction that $I_0 \models max_{\leq_{sl}}(D)$. Since D is perfectly linked with respect to T, there exists a clause $E \in T$ such that $max_{\leq_{sl}}(D) =$

$\overline{max_{\leq_{sl}}(E)}$. This is contradictory to the condition that D cannot be ordered-resolved with any clause in T. □

A consequence of this lemma is that the D in the premise cannot share the same maximum literal with any clause in T.

Corollary 35. *If*

1. *T is a perfectly linked set of ground clauses and*
2. *D is a perfectly linked ground clause with respect to T and not ordered resolvable with any ground clause in T,*

then there is no clause E in $T\backslash\{D\}$ such that $max_{\leq_{sl}}(D) = max_{\leq_{sl}}(E)$.

Proof. By Lemma 34,

$$I_0 \quad \not\models \quad max_{\leq_{sl}}(D) \tag{3.4.9}$$

By Clause 2 of Definition 29 and (3.4.9), there is no clause E in $T\backslash\{D\}$ such that $max_{\leq_{sl}}(D) = max_{\leq_{sl}}(E)$ □

Now, we switch back to the context of Definition 18. The following lemma shows that m generates a model where either L or $\overline{L}$ is not a member of the generated model.

Corollary 36. *If T is a perfectly linked set, then for every pair (E, F) of distinct ground clauses in T, $max_{\leq_{sl}}(E) \neq \overline{max_{\leq_{sl}}(F)}$.*

Proof. Suppose towards a contradiction that there exist ground clauses E and F in T_k for some index k, such that $max_{\leq_{sl}}(E) = \overline{max_{\leq_{sl}}(F)}$. Both $I_0 \models max_{\leq_{sl}}(F)$ and $I_0 \models \overline{max_{\leq_{sl}}(E)}$, which is a contradiction. □

Next, we show that there is no redundancy in our model.

Corollary 37. *If T is a perfectly linked set, then for every pair (E, F) of distinct ground clauses in T, $max_{\leq_{sl}}(E) \neq max_{\leq_{sl}}(F)$.*

Proof. By Clause 2 of Definition 29 and the "perfect linking" property of T, there exists no clause $E' \in T_l$ such that $max_{\leq_{sl}}(E) = max_{\leq_{sl}}(E')$. Since $F \in T$, $max_{\leq_{sl}}(E) \neq max_{\leq_{sl}}(F)$. □

3.4 Properties of *delete* in Definition 27

Having proved the key properties of our models, we now turn to *delete*. The purpose of *delete* is to restore the "perfectly linking" property of the set $T_k \cup \{D_{k+1}\}$ for every possible index k, so that we can generate a model that has the foregoing properties.

First, we show that $A_{k,s}$ captures the "imperfection" of $U_{k,s}$ for every possible pair of indices k and s.

Lemma 38. *If*

1. *$U_{k,0}$ is perfectly linked,*

2. *D'_k is perfectly linked with respect to $U_{k,0}$, and*

3. *L is a literal in $U_{k,s} \cup \{D'_k\}$ that violates either clause of Definition 29,*

then $L \in A_{k,s}$ for every possible pair of indices k and s.

Proof. We prove this lemma by induction on all possible indices s.

Basis step.

First, we show that D'_k is perfectly linked with respect to $U_{k,0} \cup \{D'_k\}$.

Clause 1 of Definition 29. If $I_0 \models L \in D'_k$, then by the "perfect linking" property of D'_k with respect with $U_{k,0}$, there is exactly one clause E in $U_{k,0}$ such that $L =$

$\overline{max_{\leq_{sl}}(E)}$. We only need to show that L cannot be $\overline{max_{\leq_{sl}}(D'_k)}$. We prove this by contradiction. Suppose towards a contradiction that $L = \overline{max_{\leq_{sl}}(D'_k)}$. Then both L and $\overline{L}$ are in D'_k, which is contradictory to Lemma 31.

Clause 2 of Definition 29. By Corollary 33, D'_k is perfectly linked with respect to $U_{k,0} = U_{k,0} \cup \{D'_k\} \backslash \{D'_k\}$.

Second, we show that clauses in $U_{k,0}$ which do not contain literals in $A_{k,0}$ are perfectly linked with respect to $U_{k,0} \cup \{D'_k\}$.

Clause 1 of Definition 29. Since $U_{k,0}$ is perfectly linked, for every literal L in every clause E' in $U_{k,0}$, there is exactly one clause E in $U_{k,0}$ such that $L = \overline{max_{\leq_{sl}}(E)}$. By Corollary 35, D'_k cannot share the same maximum literal with any clause E in $U_{k,0} \backslash \{D'_k\}$. Therefore, the clause E remains the only clause such that $L = \overline{max_{\leq_{sl}}(E)}$ in set $U_{k,0} \backslash \{D'_k\} \cup \{D'_k\} = U_{k,0} \cup \{D'_k\}$.

Clause 2 in Definition 29 may not hold. Since $U_{k,0}$ is perfectly linked, the only possible case that Clause 2 may not hold is when a clause E' in $U_{k,0}$ contains $max_{\leq_{sl}}(D'_k)$. But since $max_{\leq_{sl}}(D'_k) \in A_{k,0}$, we have proved the basis step.

Induction step. Our induction hypothesis (IH) is that if L is a literal in $U_{k,s} \cup \{D'_k\}$ that violates either clause in Definition 29, then $L \in A_{k,s}$. By construction, $U_{k,s+1} \subset U_{k,s}$. By (IH), we only need to consider new violations caused by the difference between $U_{k,s+1}$ and $U_{k,s}$.

Clause 1 of Definition 29. Since $U_{k,s+1} \subset U_{k,s}$, there may be more violations of Clause 1 in Definition 29 in $U_{k,s+1}$ than $U_{k,s}$, since $U_{k,s+1}$ does not contain $E_{k,s+1}$ which $U_{k,s}$ contains. The only possible case of new violation is when a clause E' in $U_{k,s+1}$ contains $\overline{max_{\leq_{sl}}(E_{k,s+1})}$. But since $\overline{max_{\leq_{sl}}(E_{k,s+1})} \in A_{k,s+1}$, we have proved the induction step for Clause 1.

Clause 2 of Definition 29. Since $U_{k,s+1} \subset U_{k,s}$, there are no more violations of Clause 2 of Definition 29 in $U_{k,s+1}$ than in $U_{k,s}$. □

Using this lemma, we can show that the *delete* function in Definition (27) indeed restores the "perfect linking" property.

Lemma 39. *If*

1. *$U_{k,0}$ is perfectly linked,*

2. *D'_k is perfectly linked with respect to $U_{k,0}$, and*

3. *the construction of the series $U_{k,s}$ stops at the $(i+1)$st iteration,*

then $U_{k,i} \cup \{D'_k\}$ is perfectly linked for every possible pair of indices k and s.

Proof. Since $U_{k,s}$ stops at the $(i+1)$st iteration, we know that there is no clause in $U_{k,i}$ that contains any literal in $A_{k,i}$. By Lemma 38, there are no more violations of the "perfect linking" property in $U_{k,i} \cup \{D'_k\}$. □

Corollary 40. *Given our m function in Definition 24 and our simp function in Definition 28, T_k in Definition* (18) *is perfectly linked for every possible index k.*

Proof. By induction on all possible indices k.

Basis step.$T_0 = \emptyset$ is perfectly linked.

Induction step. Our induction hypothesis (IH) is that T_k is perfectly linked. Since $(I_0, M_k) \not\models D_k$, for every literal L in D_{k+1}, if $I_0 \models L$, then there is a clause E in T_k such that $L = \overline{max_{\leq_{sl}}(E)}$; if $I_0 \not\models L$, then there is no clause E in T_k such that $L = max_{\leq_{sl}}(E)$. Therefore, D_{k+1} is perfectly linked with respect to T_k. By Corollary 33, D'_{k+1} is also perfectly linked with respect to T_k. By the finiteness of T_k, the construction of $U_{k,s}$ will eventually stop. By Lemma 39, $T_{k+1} = simp(T_k, D_{k+1})$ is perfectly linked. □

3.4 Proof of Proposition 20

We use the following definitions in our proof of Proposition 20.

Definition 41. The deletion set Del_k generated from $simp(T_{k-1}, D_k)$ in Definition 28 is defined as $\{E_{k,1}, \ldots, E_{k,n}\}$ where n is the maximum integer such that E_n exists.

The deletion set contains all clauses that are in T_{k-1} but not in $simp(T_{k-1}, D_k)$.

Definition 42. The addition set Add_k generated from $simp(T_{k-1}, D_k)$ in Definition 28 is defined as $\{D'_k\}$.

The addition set contains all clauses (only one in our case) that are not in T_{k-1} but are in $simp(T_{k-1}, D_k)$. One property of these two sets is that the newly added clause cannot be deleted.

Lemma 43. *$Del_k \cap Add_k = \emptyset$.*

Proof. Because $Del_k \subset T_{k-1}$, we only need to show that $Add_k \cap T_{k-1} = \emptyset$, i.e., $D'_k \notin T_{k-1}$. By Lemma 34, $I_0 \not\models max_{\leq_{sl}}(D'_k)$. By Clause 2 of Definition 29, there is no clause in T_{k-1} that contains $max_{\leq_{sl}}(D'_k)$. Therefore, $D'_k \notin T_{k-1}$. □

The following lemma shows that all deleted clauses are larger than the added clauses.

Lemma 44. *For every ground clause E in Del_k, $max_{\leq_{sl}}(D'_k) <_{sl} max_{\leq_{sl}}(E)$.*

Proof. Prove $max_{\leq_{sl}}(D'_k) <_{sl} max_{\leq_{sl}}(E_{k,s})$ for all $s \in \{1, \ldots, i\}$ by induction.

Basis step. $E_{k,1} \cap A_0 \neq \emptyset$. Since $max_{\leq_{sl}}(D'_k)$ is the only member of A_0, it must be that

$$max_{\leq_{sl}}(D'_k) \quad \in \quad E_{k,1} \tag{3.4.10}$$

By Corollary 35,

$$max_{\leq_{sl}}(D'_k) \quad \neq \quad max_{\leq_{sl}}(E_{k,1}) \tag{3.4.11}$$

By the property of D'_k that it cannot be ordered-resolved with any clause in $U_{k,0}$ and $U_{k,1} \subset U_{k,0}$,

$$max_{\leq_{sl}}(D'_k) \quad \neq \quad \overline{max_{\leq_{sl}}(E_{k,1})} \tag{3.4.12}$$

By (3.4.10), (3.4.11), and (3.4.12), $max_{\leq_{sl}}(D'_k) <_{sl} max_{\leq_{sl}}(E_{k,1})$.

Induction step. Our induction hypothesis (IH) is that for every $l \leq s$, $max_{\leq_{sl}}(D'_k)$ $<_{sl}$ $max_{\leq_{sl}}(E_{k,l})$. Since $E_{k,s+1} \cap A_{k,s} \neq \emptyset$. $E_{k,s+1}$ either contains $max_{\leq_{sl}}(D'_k)$ or $\overline{max_{\leq_{sl}}(E_{k,l})}$ for some $l \leq s$.

Case 1. $E_{k,s+1}$ contains $max_{\leq_{sl}}(D'_k)$. By a similar argument as that in the basis step, we have $max_{\leq_{sl}}(D'_k) <_{sl} max_{\leq_{sl}}(E_{k,1})$.

Case 2. $E_{k,s+1}$ contains $\overline{max_{\leq_{sl}}(E_{k,l})}$ for some $l \leq s$. By a similar argument as that in the basis step, we have

$$\overline{max_{\leq_{sl}}(E_{k,l})} \quad <_{sl} \quad max_{\leq_{sl}}(E_{k,s+1}) \tag{3.4.13}$$

By (IH),

$$max_{\leq_{sl}}(D'_k) \quad <_{sl} \quad max_{\leq_{sl}}(E_{k,l}) \tag{3.4.14}$$

By definition of $\leq_{sl}$,

$$max_{\leq_{sl}}(E_{k,l}) \quad \leq_{sl} \quad \overline{max_{\leq_{sl}}(E_{k,l})} \tag{3.4.15}$$

By (3.4.13), (3.4.14), and (3.4.15), $max_{\leq_{sl}}(D'_k) <_{sl} max_{\leq_{sl}}(E_{k,s+1})$. □

Now, we show that

Lemma 45. *Proposition 20 holds for the simp in Definition 28 and m in Definition 24.*

Proof. (3.2.8) holds since AR does not introduce any new literals. (3.2.9) holds since d does not introduce any new literals. Next, we prove (3.2.7). We need to show that $M_k <_M M_{k+1}$ for every possible index k. Expanding the definitions, we obtain $min_{\leq_{sl}}\{max_{\leq_{sl}}(E) \mid E \in Add_k\} <_{sl} min_{<_{sl}}\{max_{\leq_{sl}}(E) \mid E \in Del_k\}$. Substituting the definition of Add_k and simplifying, we have $max_{\leq_{sl}}(D'_k) <_{sl} min_{\leq_{sl}}\{max_{\leq_{sl}}(E) \mid E \in Del_k\}$, which is true according to Lemma 44. □

We have already proved all the necessary properties that guarantee the soundness and completeness of incOSHL.

Theorem 46. *incOSHL is sound and complete*

Proof. By Theorem 21, Theorem 22, Lemma 25, and Lemma 45. □

3.5 Changes in incOSHL

Having described the proof procedure of incOSHL, I would like to list the main differences between incOSHL and OSHL. First, OSHL requires D_k to be minimum with respect to $\leq_{sl}$, while incOSHL relaxes the ordering requirement for D_k to be minimal with respect to $\leq_s$. The rationale of removing the lexicographic component of $\leq_{sl}$ is that the lexicographic ordering is not necessarily an intrinsic characteristic of the input clauses, and removing it makes generating D_k less computationally intensive. Second, OSHL deletes all clauses that are larger than D'_k with respect to $\leq_{sl}$ from T_k, in order to keep the set T_{k+1} for going out of order, while incOSHL does not view the series T_k of sets of ground clauses are being ordered. Instead, incOSHL uses the *simp* function to maintain the perfect linking condition. As a result, there are usually less changes from T_k to T_{k+1}. This has three impacts:

1. There may be more literals in the model, hence the need for a more efficient data structure.

2. Number of repeated instance generations is reduced.

3. The incremental version of this algorithm is more efficient.

Chapter 4

Type Inference

In this chapter, we introduce a new feature that was not present in previous OSHL work. This feature is called “type inference.”

Compilers and automated theorem provers are similar in that they both process expressions in a certain language. The difference is that compilers usually deal with programming language comprising control structures such as branching and recursion, while automatic theorem provers usually deal with first order languages. Despite the similarity, most static analysis techniques used in compilers are not directly applicable to automatic theorem provers. Static analysis using inferred types [30] is one of the most widely used techniques in programming language design and implementation. One of the roles that types play in a programming language is to eliminate expressions that lead to an error or diverge before they are evaluated. The same idea can be applied to a set of clauses: constructing types that eliminate instances that are not needed for finding a refutation proof before starting the proof search.

We call the resulting algorithm combining genOSHL with type inference typedGenOSHL. typedGenOSHL infers types from untyped clauses as a preprocessing step. The type information is used to instantiate free variables in d, which reduces the search space, while preserving the completeness of our algorithm.

To see how type inference reduces the search space, we look at a simple example. To prove that clauses $\{P(X)\}, \{\neg P(f(f(a))\}$ are unsatisfiable, we only need instances $\{P(f(f(a)))\}$ and $\{\neg P(f(f(a)))\}$. Instances such as $\{P(a)\}$ are irrelevant. In an untyped setting, if genOSHL generate smaller terms before larger terms, it will generate $\{P(a)\}$ before generating $P(f(f(a))$, since $\{P(a)\}$ has a smaller size. However, we know that $\{P(a)\}$ is irrelevant to our proof.

In a typed setting, incOSHL will first unify $\{P(f(f(a)))\}$ and $\{P(X)\}$, resulting in a most general unifier $[f(f(a))/X]$. Then it generates a grammar

$$X \ ::= \ f(f(a))$$

and generates terms based on this grammar. This way it only generates instances that are relevant.

In general, type inference generates a function

$$[\mathcal{P}(Terms)]_{dim(\mathbf{X})} \ \rightarrow \ [\mathcal{P}(Terms)]_{dim(\mathbf{X})}$$

for a set $\mathbf{X}$ of variables, from the set S of input clauses by performing unification. Then it finds the fixpoint of this function. Each component of the fixpoint is a set of terms which corresponds to the variable in the same position in $\mathbf{X}$. The variable is called a type and the set of the terms is the extension of the type. In our type inference algorithm, a variable can only have one extension. One constraint for these sets of terms is that we need to find such sets that are as small as possible (to reduce search space as much as possible) and can be efficiently enumerated. In the current solution, the sets of terms are represented finitely as functional unions of term functions.

In this chapter, to avoid excessively detailed notations, we will assume that S denotes the set of input clauses, in which clauses are renamed to not share common variables, that $\mathbf{X}$ is a vector of variables containing all variables in S, and that $\mathbf{X}$ is the subscript of all term functions and reverse term functions.

4.1 Generating the Set Equation

4.1 An Example

Before we proceed with our discussion of the type inference algorithm, we look at another example.

Example 47. Suppose that we have input clauses $\{p(X)\}$ and $\{\neg p(f(f(Y)))\}$, if we unify $p(X)$ and $p(f(f(Y)))$, then we obtain a most general unifier $[f(f(Y))/X]$. We know that if we want to generate two ground instances from these two literals such that they are complements of each other, we need to generate instances that respect the most general unifier. In particular, if we instantiate X to term s, and Y to term t, we want that $s = f(f(t))$. This means that if we limit the instantiation of X and Y to two subsets of the set of all terms, then we need to ensure that for each term t that Y can be instantiated to, there is a term s that X can be instantiated to such that $s = f(f(t))$, if possible, and vice versa.

Let $\mathbf{X} = [X, Y]$. What we want here is a $dim(\mathbf{X})$-dimensional vector **inst** of sets of terms such that, for every variable in $\mathbf{X}$, we can restrict the instantiation of that variable to the corresponding set in **inst** while keeping genOSHL complete. Our goal of type inference is to find such a vector **inst** and find a finite representation of it.

Using notations introduced in Section 2.5, we can write down the idea we discussed in the previous paragraph as a system of set equations:

$$\overrightarrow{f(f(Y))_Y}(\pi_2(\mathbf{inst})) = \pi_1(\mathbf{inst}) \tag{4.1.1}$$

$$\overleftarrow{f(f(Y))_Y}(\pi_1(\mathbf{inst})) = \pi_2(\mathbf{inst}) \tag{4.1.2}$$

We can combine equations (4.1.1) and (4.1.2) into one equation:

$$[\overrightarrow{f(f(Y))_Y} \circ \pi_2, \overleftarrow{f(f(Y))_Y} \circ \pi_1](\mathbf{inst}) = \mathbf{inst} \tag{4.1.3}$$

In this example, our goal of type inference can now be stated as finding the fixpoint of $[\overrightarrow{f(f(Y))_Y} \circ \pi_2, \overleftarrow{f(f(Y))_Y} \circ \pi_1]$. An obvious fixpoint is $[\emptyset, \emptyset]$ which is also obviously not what we want here. We want a nonempty fixpoint. In order to ensure this, we expand (4.1.3) to the following.

$$[\overrightarrow{f(f(Y))_Y} \circ \pi_2 \sqcup \overrightarrow{c_{\mathbf{X}}}, \overleftarrow{f(f(Y))_Y} \circ \pi_1 \sqcup \overrightarrow{c_{\mathbf{X}}}](\mathbf{inst}) = \mathbf{inst} \tag{4.1.4}$$

where c is a fixed constant. Intuitively, we added (at least) a constant c in each component of **inst** so that the solution is nonempty.

Now we can take the smallest fixpoint. In this example, we can easily see that it is $\mathbf{inst} = [\{f(f(c), c\}, \{c\}]$.

In general, a fixpoint may contain infinite sets. We need to find a finite way to represent a fixpoint. We will use formal grammars to do this.

4.1 The Algorithm

In this subsection, we will describe an algorithm for generating a set equation that captures the ideas discussed in the previous subsection.

4.1 Generating $\mu(S)$

We construct the following sets. Given a set S of clauses, the set $\mu(S)$ is defined as follows:

$$\mu(S) = \{mgu_{reg}(L, \overline{N}) \mid L, N \in \bigcup S\}$$

$\mu(S)$ is the set of all most general unifiers of two literals L and $\overline{N}$, where both L and N appear in S. The function mgu_{reg} generates a most general unifier that satisfies the following regularity requirement:

Definition 48. A most general unifier σ of two terms s and t is regular if and only if

1. For every variable $X \in \mathbf{v}(s)$ or $X \in \mathbf{v}(t)$, if $\sigma(X) \neq X$, then
   (a) there does not exist variable Y such that $X \in \mathbf{v}(\sigma(Y))$.
   (b) for every variable $Y \in \mathbf{v}(\sigma(X))$, $Y \in \mathbf{v}(s)$ or $Y \in \mathbf{v}(t)$.
2. For every variable X that does not appear in s or t, $\sigma(X) = X$.

The following example shows some examples of regular and non-regular most general unifiers.

Example 49. Suppose we have literals $g(X, f(Y))$ and $g(f(Z), W)$. A regular most general unifier is $[f(Z)/X, f(Y)/W]$. Condition 1(a) in Definition 48 eliminates most general unifier $[f(X)/X, X/Z, f(Y)/W]$ from being regular. Condition 1(b) eliminates most general unifier $[f(Z')/X, Z'/Z, f(Y)/W]$ from being regular.

Condition 2 eliminates most general unifier $[f(Z)/X, f(Y)/W, X'/Z']$ from being regular.

Irregular most general unifiers will be problematic in our type inference algorithm because they cause types to be mixed up and introduce undefined types in our algorithm. Therefore, we only consider regular most general unifiers. If two terms are unifiable, there always exists a regular most general unifier.

4.1 Generating $pair(\mu(S))$

To make the presentation more concise, we construct "pairs" from most general unifiers. Given a substitution σ, the set $pair(\sigma)$ is defined as follows[1]:

$$pair(\sigma) = \{\langle X, t\rangle \mid \sigma(X) = t \text{ and } X \neq t\}$$

Example 50. $pair([f(f(a))/X, g(a, Z)/Y]) = \{\langle X, f(f(a))\rangle, \langle Y, g(a, Z)\rangle\}$.

We extend *pair* to sets of substitutions in a natural manner. Give a set A of substitutions,

$$pair(A) = \bigcup_{\sigma \in A} pair(\sigma)$$

4.1 Generating $\mathbf{F}(pair(\mu(S)))$

We define a function $\mathbf{F} : \mathcal{P}(V \times Terms(F, V)) \to [\mathcal{P}(Form_1 \cup Form_2)]_{dim(\mathbf{X})}$, where V is the set of all variables, F is the set of all function symbols, $Form_1$ is the set of all term functions of Form 1 from Definition 9, and $Form_2$ is the set of all functions of Form 2 from Definition 9.

[1]This step essentially breaks down substitutions to pairs. Pairs make it explicit which variables we need to deal with, which makes it easier to present the formalism for the following steps.

Given a pair $\langle X, t\rangle$, let j be the index of X in $\mathbf{X}$. We define vector $\mathbf{F}(\langle X, t\rangle)$ as a vector identical to $\mathbf{e}$ (recall that $\mathbf{e}$ denotes a vector of empty sets) except

1. $\pi_j(\mathbf{F}(\langle X, t\rangle)) = \{\overrightarrow{t}\}$
2. for every variable Y in $\mathbf{v}(t)$, $\pi_i(\mathbf{F}(\langle X, t\rangle)) = \{\pi_i \circ \overleftarrow{t} \circ \pi_j\}$, where i is the index of Y in $\mathbf{X}$.

Example 51. If $\mathbf{X} = [X, Y, Z]$, then $\mathbf{F}(\langle Y, g(a, Z)\rangle) = [\emptyset, \{\overrightarrow{g(a, Z)}\}, \{\pi_3 \circ \overleftarrow{g(a, Z)} \circ \pi_2\}]$. The first component comes from the vector $\mathbf{e}$; the second component comes from 1; and the third component comes from 2.

We extend $\mathbf{F}$ to a set P of pairs as follows:

$$\mathbf{F}(P) \quad = \quad \bigcup_{\langle X,t\rangle \in P} \mathbf{F}(\langle X, t\rangle)$$

Example 52. If $\mathbf{X} = [X, Y, Z]$, then

$$\begin{aligned} & \mathbf{F}(\{\langle X, f(f(a))\rangle, \langle Y, g(a, Z)\rangle\}) \\ = \; & \mathbf{F}(\langle X, f(f(a))\rangle) \cup \mathbf{F}(\langle Y, g(a, Z)\rangle) \\ = \; & [\{\overrightarrow{f(f(a))}\}, \emptyset, \emptyset] \cup [\emptyset, \{\overrightarrow{g(a, Z)}\}, \{\pi_3 \circ \overleftarrow{g(a, Z)} \circ \pi_2\}] \\ = \; & [\{\overrightarrow{f(f(a))}\}, \{\overrightarrow{g(a, Z)}\}, \{\pi_3 \circ \overleftarrow{g(a, Z)} \circ \pi_2\}] \end{aligned}$$

The vector that we are interested in is $\mathbf{F}(pair(\mu(S)))$. We will generate all terms for all variables based on this vector. However, there is still a problem. If we look at the vector generated in Example 52, then we can see that $\overleftarrow{g(a, Z)}$ generates smaller terms from larger terms. In our search algorithm for a proof, we would like to start from smaller terms and go up to larger terms monotonically. Therefore, we would like to find a way to remove reverse term functions from the generated vector. Simply removing them will result in a vector that is too coarse to generate all useful terms, as shown in the following example

Example 53. If $\mathbf{X} = [X, Y]$, and we have clauses $S = \{\{p(X), q(X)\}, \{\neg P(f(a))\}, \{\neg Q(f(Y))\}\}$, then we know that in order to find a proof, we need to instantiate Y to a. We have

$$\mathbf{F}(pair(\mu(S))) \quad = \quad [\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \{\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1\}]$$

However, simply removing $\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1$ from this vector results in vector $[\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \emptyset]$, which does not generate anything for Y.

What we need is to refine the generated sets.

4.1 Generating $\mathbf{G}(pair(\mu(S)))$

We introduce a construction $\mathbf{E}$ as follows

$$\begin{aligned}
\mathbf{E}_0(\mathbf{A}) &= \mathbf{A} \\
\mathbf{E}_{k+1}(\mathbf{A}) &= \mathbf{E}_k(\mathbf{A}) \cup \mathbf{e}[\pi_q(\mathbf{A}) \cup \{\overrightarrow{Y}\}/p][\pi_p(\mathbf{A}) \cup \{\overrightarrow{Y}\}/q] \text{ where} \\
& \quad X = \pi_p(\mathbf{X}),\ Y = \pi_q(\mathbf{X}), \text{ and } \overrightarrow{X} \in \pi_q(\mathbf{A}) \\
\mathbf{E}(\mathbf{A}) &= \bigcup_{k=0}^{\infty} \mathbf{E}_k(\mathbf{A})
\end{aligned}$$

The basic idea of $\mathbf{E}$ is: If $\overrightarrow{X} \in \pi_q(\mathbf{A})$, then it means that the variable at index q should generate the same terms as the variable X; therefore, we merge the set at index q in $\mathbf{A}$ and the set at index p in $\mathbf{A}$.

Example 54. If $\mathbf{X} = [X, Y]$. We have

$$\begin{aligned} & \mathbf{E}([\{\overrightarrow{f(a)}, \overrightarrow{Y}\}, \{\pi_2 \circ \overleftarrow{Y} \circ \pi_1\}, \{\overrightarrow{f(f(b))}\}]) \\ = \quad & [\{\overrightarrow{f(a)}, \overrightarrow{X}, \overrightarrow{Y}, \pi_2 \circ \overleftarrow{Y} \circ \pi_1\}, \{\overrightarrow{f(a)}, \overrightarrow{X}, \overrightarrow{Y}, \pi_2 \circ \overleftarrow{Y} \circ \pi_1\}, \{\overrightarrow{f(f(b))}]\end{aligned}$$

Here, we merge set at index 1 with set at index 2 because $\overrightarrow{Y}$ belongs to the set at index 1. After merging, both sets contain $\overrightarrow{X}$, $\overrightarrow{Y}$, and all other functions from the original sets.

We introduce a construction $\mathbf{G}$ that refines $\mathbf{F}$.

Definition 55. We obtain a set $\mathbf{G}(P)$ from a set $\mathbf{F}(P)$ as follows.

$$\begin{aligned} \mathbf{G}_0(P) \quad &= \quad \mathbf{F}(P) \\ \mathbf{G}_{k+1}(P) \quad &= \quad \mathbf{E}(\mathbf{G}_k(P) \cup \mathbf{F}(pair(\sigma))) \text{ where there is} \\ & \quad\quad \text{a variable } X \text{ with index } i \text{ in } \mathbf{X}, \\ & \quad\quad \text{a function } \pi_i \circ \overleftarrow{s} \circ \pi_j \in \pi_i(\mathbf{G}_k(P)), \\ & \quad\quad \text{a function } \overrightarrow{t} \in \pi_j(\mathbf{G}_k(P)) \text{ such that} \\ & \quad\quad mgu_{reg}(s, t) = \sigma \\ \mathbf{G}(P) \quad &= \quad \bigcup_{k=0}^{\infty} \mathbf{G}_k(P) \end{aligned}$$

Intuitively, this construction expands term functions into reverse term functions: each iteration in this construction expands $\overrightarrow{t}$ into the reverse term function $\overleftarrow{s}$ and generates new term functions and reverse term functions.

Example 56. Following Example 53, we have

$$\mathbf{F}(pair(\mu(S))) \quad = \quad [\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \{\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1\}]$$

If we apply construction $\mathbf{G}$, then we have

$$\mathbf{G}_0(pair(\mu(S))) \quad = \quad [\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \{\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1\}]$$

To construct $\mathbf{G}_1(pair(\mu(S)))$, we choose a variable Y with index 2, a function $\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1$, and a function $\overrightarrow{f(a)}$. We have $mgu_{reg}(f(Y), f(a)) = [a/Y]$ and

$$\mathbf{F}(pair([a/Y])) \quad = \quad [\emptyset, \{\overrightarrow{a}\}]$$

Therefore,

$$\begin{aligned}
\mathbf{G}_1(pair(\mu(S))) \quad &= \quad \mathbf{E}(\mathbf{G}_0(pair(\mu(S))) \cup \mathbf{F}(pair([a/Y]))) \\
&= \quad \mathbf{E}([\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \{\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1\}] \cup [\emptyset, \{\overrightarrow{a}\}]) \\
&= \quad [\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \{\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1, \overrightarrow{a}\}]
\end{aligned}$$

Here, we added $\overrightarrow{a}$ to the vector. Therefore, we can safely delete $\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1$ from it, and we are still able to generate term a when instantiating Y. If we keep performing this construction, eventually we have

$$\mathbf{G}(pair(\mu(S))) \quad = \quad [\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}, \{\pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1, \overrightarrow{a}, \overrightarrow{Y}\}]$$

4.1 Adding A Constant to $\mathbf{G}(pair(\mu(S)))$

We choose a fixed constant c, add $\overrightarrow{c}$ to each component of $\mathbf{G}(pair(\mu(S)))$ and combine all functions in $\pi_i(\mathbf{G}(pair(\mu(S))))$ into one using the $\bigsqcup$ operator for $1 \leq i \leq dim(\mathbf{X})$. We denote the resulting vector by $\bigsqcup(\mathbf{G}(pair(\mu(S))) \cup [\overrightarrow{c}]_{i=1}^{dim(\mathbf{X})})$, which is a vector of functions of the form $\bigsqcup_{k=1}^{n} f_k$, where each f_k is a function of the form $\overrightarrow{t}$ or $\pi_i \circ \overleftarrow{t} \circ \pi_j$.

Example 57. Following Example 56, we choose the constant to be a, and our vector looks like$[\overrightarrow{f(a)} \sqcup \overrightarrow{f(Y)} \sqcup \overrightarrow{a}, \pi_2 \circ \overleftarrow{f(Y)} \circ \pi_1 \sqcup \overrightarrow{a} \sqcup \overrightarrow{Y}]$.

For some set that generates at least one term, adding a constant is not necessary. For example, if we have a set $\{\overrightarrow{f(a)}, \overrightarrow{f(Y)}\}$ in our vector, then we do not need to add a constant to that set. In our theorem prover implementation, we implemented this optimization. But for the presentation of the general algorithm, we assume that a constant is added to every set in our vector for simplicity.

4.1 Generating An Equation

To make the representation succinct, we denote the vector $\bigsqcup \mathbf{G}(pair(\mu(S)) \cup [\overrightarrow{c}]_{i=1}^{dim(\mathbf{X})})$ generated from the implicit set S of input clauses by $\mathbf{f}$. The fixpoint of $\mathbf{f}$ is a solution to the following equation:

$$\mathbf{f}(\mathbf{inst}) \quad = \quad \mathbf{inst} \tag{4.1.5}$$

where **inst** is a vector of sets of terms that we want to solve.

4.2 Solving the Equation

4.2 Finding the Least Fixpoint

We want to find a solution to (4.1.5) that is as small as possible. Ideally, we want to find the least fixpoint of 4.1.5. This fixpoint $\mathbf{I}^*$ can be constructed as the limit of the following sequence (recall that $\mathbf{e} = [\emptyset, \ldots, \emptyset]$)

$$\begin{aligned} \mathbf{I}_0 &= \mathbf{e} \\ \mathbf{I}_{k+1} &= \mathbf{f}(\mathbf{I}_k) \cup \mathbf{I}_k \\ \mathbf{I}^* &= \bigcup_{k=1}^{\infty} \mathbf{I}_k \end{aligned}$$

Given any vector $\mathbf{h}$ of functions, we define a construction $\mathbf{h}^*$ (where $\mathbf{h}$ is the parameter of this construction) as the limit of the following sequence

$$\begin{aligned} \mathbf{h}^{(0)} &= \mathbf{id} \\ \mathbf{h}^{(k+1)} &= \mathbf{h} \circ \mathbf{h}^{(k)} \sqcup \mathbf{h}^{(k)} \\ \mathbf{h}^* &= \bigsqcup_{k=0}^{\infty} \mathbf{h}^{(k)} \end{aligned}$$

Indeed, $\mathbf{f}^*$ and $\mathbf{I}^*$ are equivalent in the following sense.

Lemma 58. $\mathbf{I}^* = \mathbf{f}^*(\mathbf{e})$.

Proof. Prove by induction that $\mathbf{I}_k = \mathbf{f}^{(k)}(\mathbf{e})$.

Basis step. $\mathbf{I}_0 = \mathbf{e} = \mathbf{id}(\mathbf{e}) = \mathbf{f}^{(0)}(\mathbf{e})$.

Induction step. Our induction hypothesis (IH) is $\mathbf{I}_k = \mathbf{f}^{(k)}(\mathbf{e})$.

$$\begin{aligned} \mathbf{I}_{k+1} &= \mathbf{f}(\mathbf{I}_k) \cup \mathbf{I}_k \\ &= \mathbf{f}(\mathbf{f}^{(k)}(\mathbf{e})) \cup \mathbf{f}^{(k)}(\mathbf{e}) \text{ by (IH)} \\ &= (\mathbf{f} \circ \mathbf{f}^{(k)})(\mathbf{e}) \cup \mathbf{f}^{(k)}(\mathbf{e}) \\ &= (\mathbf{f} \circ \mathbf{f}^{(k)} \sqcup \mathbf{f}^{(k)})(\mathbf{e}) \\ &= \mathbf{f}^{(k+1)}(\mathbf{e}) \end{aligned}$$

□

The advantage of using $\mathbf{f}^*$ to represent the fixpoint is that we can perform symbolic computations over the functions without having to generate the actual sets of terms.

4.2 Simplifying the Least Fixpoint

The goal of simplification is to ensure that the terms in the extension of types can be generated inductively. We use the word "inductively" in the following sense: larger terms are generated from smaller terms. Recall that the coefficient $\mathbf{f}$ to our set equation (4.1.5) is a vector of functions of the form $\bigsqcup_{k=1}^{n} f_k$, where each f_k is a function of the form $\overrightarrow{t}$ or $\pi_p \circ \overleftarrow{t} \circ \pi_q$. Among these functions, term functions are inductive, but reverse term functions are not since they produce smaller terms from larger terms. The goal of this section is to show that $\mathbf{f}^{\rightarrow *} = \mathbf{f}^*$.

4.2 Bounding Expressions in $\mathbf{f}^*$

First, we establish an upper bound for expression of the form $\pi_p \circ \overleftarrow{s} \circ \overrightarrow{t}$, where s and t are terms, in terms of term functions that appear in $\mathbf{f}^{\rightarrow}$. This is important since $\mathbf{f}^*$ contains subexpressions of this form but $(\mathbf{f}^{\rightarrow})^*$ does not. We will establish this upper bound through the following lemmas.

The first lemma gives an explicit representation of the set produced by applying $\overleftarrow{s} \circ \overrightarrow{t}$ to some set of vectors of terms.

Lemma 59. *Given*

1. *two terms s_1 and s_2 with regular mgu σ,*

2. *a vector $\mathbf{X}$ of variables containing all variables in s_1 and s_2, and*

3. a set A of $dim(\mathbf{X})$-dimensional vectors of terms,

we have

$$\{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\} \quad = \quad (\overleftarrow{s_1} \circ \overrightarrow{s_2})(A) \tag{4.2.1}$$

Proof. We prove that $\{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\} \subset (\overleftarrow{s_1} \circ \overrightarrow{s_2})(A)$ and $(\overleftarrow{s_1} \circ \overrightarrow{s_2})(A) \subset \{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\}$.

$\{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\} \subset (\overleftarrow{s_1} \circ \overrightarrow{s_2})(A)$. For every

$$\mathbf{t} \quad \in \quad \{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\}$$

we show that

$$\mathbf{t} \quad \in \quad (\overleftarrow{s_1} \circ \overrightarrow{s_2})(\mathbf{t}')$$

We have

$$\begin{aligned} (\overleftarrow{s_1} \circ \overrightarrow{s_2})(\mathbf{t}') &= \overleftarrow{s_1}(\overrightarrow{s_2}(\mathbf{t}')) \\ &= \overleftarrow{s_1}(\overrightarrow{s_1}(\mathbf{t})) \end{aligned}$$

Since $\mathbf{t} \in \overleftarrow{s_1}(\overrightarrow{s_1}(\mathbf{t}))$, we have

$$\mathbf{t} \quad \in \quad (\overleftarrow{s_1} \circ \overrightarrow{s_2})(\mathbf{t}')$$

$(\overleftarrow{s_1} \circ \overrightarrow{s_2})(A) \subset \{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\}$. We need to show that for every $\mathbf{t}'' \in A$ and every

$$\mathbf{t} \quad \in \quad (\overleftarrow{s_1} \circ \overrightarrow{s_2})(\mathbf{t}'') \tag{4.2.2}$$

we have

$$\mathbf{t} \in \{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\} \tag{4.2.3}$$

By (4.2.2),

$$\begin{aligned} \overrightarrow{s_1}(\mathbf{t}) &\in \overrightarrow{s_1}((\overleftarrow{s_1} \circ \overrightarrow{s_2})(\mathbf{t}'')) \\ &= \overrightarrow{s_1}(\overleftarrow{s_1}(\overrightarrow{s_2}(\mathbf{t}''))) \end{aligned} \tag{4.2.4}$$

Case 1. If $\overrightarrow{s_2}(\mathbf{t}'')$ is not an instance of s_1, then

$$\overleftarrow{s_1}(\overrightarrow{s_2}(\mathbf{t}'')) = \emptyset$$

Therefore, "for all (4.2.2), (4.2.3)" is trivially true because we can find any $\mathbf{t}$ that satisfies (4.2.2).

Case 2. If $\overrightarrow{s_2}(\mathbf{t}'')$ is an instance of s_1, then

$$\overrightarrow{s_1}(\overleftarrow{s_1}(\overrightarrow{s_2}(\mathbf{t}''))) = \{\overrightarrow{s_2}(\mathbf{t}'')\} \tag{4.2.5}$$

By (4.2.4) and (4.2.5),

$$\overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'')$$

Since $\mathbf{t}'' \in A$,

$$\mathbf{t} \in \{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\}$$

□

The second lemma gives a bound to the set in the previous lemma in terms of some term functions.

Lemma 60. *Given*

1. *two variable-disjoint terms s_1 and s_2 with regular mgu σ,*

2. *a vector $\mathbf{X}$ of variables containing all variables in s_1 and s_2, and*

3. a set A of vectors of terms with length $dim(\mathbf{X})$,

4. *a variable X in s_1 with index p in $\mathbf{X}$,*

then there exists an integer K such that

$$\pi_p(\{\mathbf{t} \mid \overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}'), \mathbf{t}' \in A\}) \quad \subset \quad \pi_p(\mathbf{f}_\sigma^{(K)}(A)) \tag{4.2.6}$$

where $\mathbf{f}_\sigma = \mathbf{G}(pair(\sigma))$.

Proof. First, note that there are four groups of variables here:

1. Variable Y in s_1 such that $\sigma(Y) = Y$

2. Variable Y in s_2 such that $\sigma(Y) = Y$

3. Variable Y in s_1 such that $\sigma(Y) \neq Y$

4. Variable Y in s_2 such that $\sigma(Y) \neq Y$

It can be easily seen that Group 1 and Group 3 are disjoint and Group 2 and Group 4 are disjoint. We define a directed graph $\langle V, E\rangle$, where

1. V is the set of all variables in s_1 and s_2, and

2. E is the set defined as follows: for every variable Y,

(a) If Y belongs to Group 1, then there is an edge $\langle Y, Z \rangle \in E$ for all Z in Group 4 such that Y appears in $\sigma(Z)$. This edge corresponds to reverse term function $\pi_r \circ \overleftarrow{\sigma(Z)} \circ \pi_q$ where r is the index of Y in $\mathbf{X}$ and q is the index of Z in $\mathbf{X}$.

(b) If Y belongs to Group 3, then there is an edge $\langle Y, Z \rangle \in E$ for all Z that appears in $\sigma(Y)$. These edges correspond to term function $\overrightarrow{\sigma(Y)}$.

Fact 1: there is no cycle in this graph. Suppose towards a contradiction that this is not the case. Since in our construction of the graph, 2(a) generates an edge from Group 1 to Group 4 and we do not generate any edge going out from Group 4, 2(a) does not generate any cycle. Therefore, only 2(b) may generate cycles, i.e., there is a list $X_1, \ldots, X_n$ of variables such that

- If $n = 1$, X_1 appears in $\sigma(X_1)$ and $X_1 \neq \sigma(X_1)$. This is contradictory to Definition 48.
- If $n = 2$, X_2 appears in $\sigma(X_1)$, ..., X_n appears in $\sigma(X_{n-1})$ and X_1 appears in $\sigma(X_n)$. Therefore, X_2 appears in $\sigma(X_1)$ and $X_2 \neq \sigma(X_2)$. This is also contradictory to Definition 48.

Fact 2: for every variable Y in Group 1, there is a variable Z such that there is an edge $\langle Y, Z \rangle$. We prove that there must be a variable Z in Group 4 such that Y appears in $\sigma(Z)$. Suppose towards a contradiction that such Z does not exist. Then, Y is a variable that only appears in $\sigma(s_1)$ but not $\sigma(s_2)$ which is a contradiction.

Fact 3: for every variable Y in Group 3, either $\sigma(Y)$ is ground or there is a variable Z such that there is an edge $\langle Y, Z \rangle$. It is trivial to see that this is true.

In summary, for every variable Y in Group 1 or Group 3, we can trace back to variables in Group 2 or Group 4. Each edge either corresponds to a term function or a reverse term function. Therefore, we can construct all terms in σ using these functions by following the edges.

We want to prove that there exists an integer K such that for every $\mathbf{t}' \in A$, and every $\mathbf{t}$ such that $\overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}')$, $\pi_p(\mathbf{t}) \subset \pi_p(\mathbf{f}_\sigma^{(K)}(A))$. Let $t = \sigma(X)$.

Case 1. $t = X$. X belongs to Group 1. By Fact 2 and 2(a), there is a variable Z in Group 4 in $\mathbf{X}$ such that X appears in $\sigma(Z)$. Let the index of Z in $\mathbf{X}$ be q. We know that if $\overrightarrow{s_1}(\mathbf{t}) = \overrightarrow{s_2}(\mathbf{t}')$, then $\pi_p(\mathbf{t}) \in (\pi_p \circ \overleftarrow{\sigma(Z)} \circ \pi_q)(\mathbf{t}')$. Since $\pi_p \circ \overleftarrow{\sigma(Z)} \circ \pi_q$ appears in $\mathbf{f}_\sigma$. We only need $K = 1$.

Case 2. $t \neq X$. X belongs to Group 3. Now, since σ is a most general unifier, $\sigma(s_1) = \sigma(s_2)$. Let $s = \sigma(s_1)$. For every variable Y in $\mathbf{v}(s)$, by Definition 48, it belongs to either Group 1 or Group 2. Either way, it is not affect by σ. Suppose that Y belongs to Group 1. Since $[\mathbf{t}/\mathbf{X}](s_1)$ is an instance of s, there must be a substitution η such that $[\mathbf{t}/\mathbf{X}](s_1) = \eta(s)$. Therefore, $\eta(Y) = [\mathbf{t}/\mathbf{X}](Y)$. Similarly, if Y belongs to Group 2, there is a substitution η' such that $[\mathbf{t}'/\mathbf{X}](s_2) = \eta'(s)$ and $\eta'(Y) = [\mathbf{t}'/\mathbf{X}](Y)$.

Let all variables in s belonging to Group 2 be $X_1, \ldots, X_m$ and have indices $p_1, \ldots, p_m$ in $\mathbf{X}$ and all variables in s belonging to Group 1 be $Y_1, \ldots, Y_n$ and have indices $r_1, \ldots, r_n$ in $\mathbf{X}$. We can construct a new vector

$$\mathbf{t}'' \;=\; \mathbf{e}[\pi_{p_1}(\mathbf{t}')/p_1]\ldots[\pi_{p_m}(\mathbf{t}')/p_m][\pi_{r_1}(\mathbf{t})/p_1]\ldots[\pi_{r_m}(\mathbf{t})/p_m]$$

such that $[\mathbf{t}''/\mathbf{X}](s) = [\mathbf{t}/\mathbf{X}](s_1) = [\mathbf{t}'/\mathbf{X}](s_2)$. Therefore, $[\mathbf{t}/\mathbf{X}](X) = ([\mathbf{t}''/\mathbf{X}] \circ \sigma)(X)$.

$$\pi_p(\mathbf{t}) \;=\; \overrightarrow{t}(\mathbf{t}'') \qquad (4.2.7)$$

By Case 1, for $1 \leq i \leq n$, we can find a variable Z_i in Group 4 with index q_i

such that

$$\pi_{r_i}(\mathbf{t}) \;\in\; (\pi_{r_i} \circ \overleftarrow{\sigma(Z_i)} \circ \pi_{q_i})(\mathbf{t}') \tag{4.2.8}$$

By (4.2.8), we have

$$\mathbf{t}'' \;\subset\; (\mathbf{f}_\sigma \sqcup \mathbf{id})(\mathbf{t}') \tag{4.2.9}$$

Since

$$\overrightarrow{t} \;\sqsubset\; \pi_p(\mathbf{f}_\sigma) \tag{4.2.10}$$

By (4.2.7), (4.2.9), and (4.2.10), we have

$$\begin{aligned}
\pi_p(\mathbf{t}) &= \overrightarrow{t}(\mathbf{t}'') \\
&\subset (\pi_p(\mathbf{f}_\sigma))(\mathbf{t}'') \\
&\subset (\pi_p(\mathbf{f}_\sigma))((\mathbf{f}_\sigma \sqcup \mathbf{id})(\mathbf{t}')) \\
&= \pi_p((\mathbf{f}_\sigma \circ (\mathbf{f}_\sigma \sqcup \mathbf{id}))(\mathbf{t}'))
\end{aligned}$$

Therefore, we only need $K = 2$.

In summary, we only need $K = 2$ in both cases. □

Combining the previous two lemmas, we have

Lemma 61. *Given*

1. *variable-disjoint terms s_1 and s_2 with regular mgu σ,*

2. *a vector $\mathbf{X}$ of variable containing all variables in s_1 and s_2, and*

3. *a variable X and an index p such that $X \in \mathbf{v}(s_1)$ and $\pi_p(\mathbf{X}) = X$,*

then there exists an integer K such that

$$\pi_p \circ \overleftarrow{s_1} \circ \overrightarrow{s_2} \quad \sqsubset \quad \pi_p \circ \mathbf{f}_\sigma^{(K)}$$

where $\mathbf{f}_\sigma = \mathbf{G}(pair(\sigma))$.

Proof. Directly follows from Lemma 59 and Lemma 60. □

In the following subsections, we prove some of the properties of $\mathbf{f}^{\rightarrow}$. The main goal is to show that $\mathbf{f}^{\rightarrow}$ has the same limit as $\mathbf{f}$.

4.2 $(\mathbf{f}^{\rightarrow})^*(\mathbf{e})$ Is a Fixpoint of (4.1.5)

First, we show that applying $\mathbf{f}^{\leftarrow}$ to $\mathbf{f}^{\rightarrow}$ does not generate any more new terms than those generated by $(\mathbf{f}^{\rightarrow})^*$.

Lemma 62. $\mathbf{f}^{\leftarrow} \circ \mathbf{f}^{\rightarrow} \sqsubset (\mathbf{f}^{\rightarrow})^*$.

Proof. It is sufficient to show that for every function of the form $\pi_p \circ \overleftarrow{t} \circ \pi_q$ that appears in $\mathbf{f}^{\leftarrow}$ and every function of the form $\overrightarrow{s}$ that appears in $\pi_q(\mathbf{f}^{\rightarrow})$, $\pi_p \circ \overleftarrow{t} \circ \overrightarrow{s} \sqsubset \pi_p \circ (\mathbf{f}^{\rightarrow})^*$. By Lemma 61, we have

$$\begin{aligned} \pi_p \circ \overleftarrow{t} \circ \overrightarrow{s} \quad &\sqsubset \quad \pi_p \circ \mathbf{f}_\sigma^{(K)} \\ &\sqsubset \quad \pi_p \circ (\mathbf{f}^{\rightarrow})^* \end{aligned}$$

□

Next, we show that applying $\mathbf{f}^{\rightarrow}$ to $(\mathbf{f}^{\rightarrow})^*$ does not generate any more new terms.

Lemma 63. $\mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^* \sqsubset (\mathbf{f}^{\rightarrow})^*$

Proof. We can prove by induction that

$$(\mathbf{f}^{\rightarrow})^{(k')} \quad \sqsubset \quad (\mathbf{f}^{\rightarrow})^{(k)} \tag{4.2.11}$$

for all $k' < k$.

Every component of $\mathbf{f}^{\rightarrow}$ is a function of the form $\bigsqcup_{i=1}^{n} \overrightarrow{t_i}$ for some n. Without loss of generality, we consider the 1st component. By left-distributivity of $\circ$ over $\sqcup$, we only need to show that for every term function $\overrightarrow{t_i}$, we have $\overrightarrow{t_i} \circ (\mathbf{f}^{\rightarrow})^* \sqsubset \pi_1 \circ (\mathbf{f}^{\rightarrow})^*$. Given a vector $\mathbf{v}$, expanding the definition of term functions,

$(\overrightarrow{t_i} \circ (\mathbf{f}^{\rightarrow})^*)(\mathbf{v}) = \{\overrightarrow{t_i}(\mathbf{v}') \mid \mathbf{v}' \in \prod(\mathbf{f}^{\rightarrow})^*(\mathbf{v})\}$

By (4.2.11), we have for every vector $\mathbf{v}' \in \prod(\mathbf{f}^{\rightarrow})^*(\mathbf{v})$ there exists k such that $\mathbf{v}' \in \prod(\mathbf{f}^{\rightarrow})^{(k)}(\mathbf{v})$. Therefore,

$$\begin{aligned}
\overrightarrow{t_i}(\mathbf{v}') &\in \overrightarrow{t_i}((\mathbf{f}^{\rightarrow})^{(k)}(\mathbf{v})) \\
&\subset (\bigsqcup_{i=1}^{n} \overrightarrow{t_i})((\mathbf{f}^{\rightarrow})^{(k)}(\mathbf{v})) \\
&= \pi_1(\mathbf{f}^{\rightarrow})((\mathbf{f}^{\rightarrow})^{(k)}(\mathbf{v})) \\
&= \pi_1((\mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{v})) \\
&\subset \pi_1((\mathbf{f}^{\rightarrow})^{(k+1)}(\mathbf{v})) \\
&\subset \pi_1((\mathbf{f}^{\rightarrow})^*(\mathbf{v}))
\end{aligned}$$

Therefore,

$$(\overrightarrow{t_i} \circ (\mathbf{f}^{\rightarrow})^*)(\mathbf{v}) \quad \subset \quad \pi_1((\mathbf{f}^{\rightarrow})^*(\mathbf{v}))$$

Hence,

$$\overrightarrow{t_i} \circ (\mathbf{f}^{\rightarrow})^* \quad \sqsubset \quad \pi_1 \circ (\mathbf{f}^{\rightarrow})^*$$

$\square$

Next, we show that applying $(\mathbf{f}^{\rightarrow})^*$ to $(\mathbf{f}^{\rightarrow})^*$ does not generate any more new terms.

Lemma 64. $(\mathbf{f}^{\rightarrow})^* \circ (\mathbf{f}^{\rightarrow})^* \sqsubset (\mathbf{f}^{\rightarrow})^*$.

Proof.

$$\begin{aligned}
(\mathbf{f}^{\rightarrow})^* \circ (\mathbf{f}^{\rightarrow})^* &= \bigsqcup_{k=0}^{\infty} (\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^* \\
&= \bigsqcup_{k=0}^{\infty} ((\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^*)
\end{aligned}$$

We prove by induction that $(\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^* \sqsubset (\mathbf{f}^{\rightarrow})^*$.

Basis step. $k = 0$. Trivial.

Induction step. Our induction hypothesis (IH) is that for k, $(\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^* \sqsubset (\mathbf{f}^{\rightarrow})^*$. We need to show that $(\mathbf{f}^{\rightarrow})^{(k+1)} \circ (\mathbf{f}^{\rightarrow})^* \sqsubset (\mathbf{f}^{\rightarrow})^*$

$$\begin{aligned}
(\mathbf{f}^{\rightarrow})^{(k+1)} \circ (\mathbf{f}^{\rightarrow})^* &= (\mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)} \sqcup (\mathbf{f}^{\rightarrow})^{(k)}) \circ (\mathbf{f}^{\rightarrow})^* \text{ by expanding the definition} \\
&= \mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^* \sqcup (\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^* \text{ by Lemma 10} \\
&= \mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)} \circ (\mathbf{f}^{\rightarrow})^* \sqcup (\mathbf{f}^{\rightarrow})^* \text{ by (IH)} \\
&\sqsubset \mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^* \sqcup (\mathbf{f}^{\rightarrow})^* \text{ by (IH)} \\
&\sqsubset (\mathbf{f}^{\rightarrow})^* \sqcup (\mathbf{f}^{\rightarrow})^* \text{ by Lemma 63} \\
&= (\mathbf{f}^{\rightarrow})^*
\end{aligned}$$

□

Next, we show that $\circ$ is right-distributive over $\sqcup$ when $\mathbf{f}^{\leftarrow}$ is its left operand.

Lemma 65. $\mathbf{f}^{\leftarrow} \circ (\bigsqcup_{k=1}^{\infty} \mathbf{g}_k) = \bigsqcup_{k=1}^{\infty}(\mathbf{f}^{\leftarrow} \circ \mathbf{g}_k)$.

Proof. Every component of $\mathbf{f}^{\rightarrow}$ is a function of the form $\bigsqcup_{i=1}^{n}(\pi_{p_i} \circ \overleftarrow{s_i} \circ \pi_{q_i})$ for some n. Without loss of generality, we consider the 1st component.

$$\begin{aligned}
\bigsqcup_{i=1}^{n}(\pi_{p_i} \circ \overleftarrow{s_i} \circ \pi_{q_i}) \circ (\bigsqcup_{k=1}^{\infty} \mathbf{g}_k) &= \bigsqcup_{i=1}^{n}((\pi_{p_i} \circ \overleftarrow{s_i} \circ \pi_{q_i}) \circ (\bigsqcup_{k=1}^{\infty} \mathbf{g}_k)) \text{ by Lemma 10} \\
&= \bigsqcup_{i=1}^{n}\bigsqcup_{k=1}^{\infty}((\pi_{p_i} \circ \overleftarrow{s_i} \circ \pi_{q_i}) \circ \mathbf{g}_k) \text{ by Lemma 11} \\
&= \bigsqcup_{k=1}^{\infty}\bigsqcup_{i=1}^{n}((\pi_{p_i} \circ \overleftarrow{s_i} \circ \pi_{q_i}) \circ \mathbf{g}_k) \text{ by switchability of } \bigsqcup \\
&= \bigsqcup_{k=1}^{\infty}((\bigsqcup_{i=1}^{n}(\pi_{p_i} \circ \overleftarrow{s_i} \circ \pi_{q_i})) \circ \mathbf{g}_k) \text{ by Lemma 10}
\end{aligned}$$

□

Next, we show that $(\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$ is bounded by $(\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e})$.

Lemma 66. $(\mathbf{f}^{\rightarrow})^{*}(\mathbf{e}) \subset (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e})$.

Proof. We prove by induction that every $0 \leq k$,

$$((\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \quad \subset \quad (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e})$$

Basis step. $((\mathbf{f}^{\rightarrow})^{(0)})(\mathbf{e}) = \mathbf{e} \subset (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e})$.

Induction step. Our induction hypothesis (IH) is $((\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \subset (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e})$.

$$\begin{aligned} ((\mathbf{f}^{\rightarrow})^{(k+1)})(\mathbf{e}) &= (\mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{k} \sqcup (\mathbf{f}^{\rightarrow})^{k})(\mathbf{e}) \\ &= (\mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{k})(\mathbf{e}) \cup (\mathbf{f}^{\rightarrow})^{k}(\mathbf{e}) \\ &\subset (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e}) \cup (\mathbf{f}^{\rightarrow})^{k}(\mathbf{e}) \\ &\subset (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e}) \cup (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e}) \text{ by (IH)} \\ &= (\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e}) \end{aligned}$$

□

Next, we show that $(\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e})$ is bounded by $(\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$.

Lemma 67. $(\mathbf{f} \circ (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e}) \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$.

Proof. Note that

$$\begin{aligned} \mathbf{f}((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) &= (\mathbf{f}^{\leftarrow} \sqcup \mathbf{f}^{\rightarrow})((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) \\ &= \mathbf{f}^{\leftarrow}((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) \cup \mathbf{f}^{\rightarrow}((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) \end{aligned}$$

By Lemma 63, $\mathbf{f}^{\rightarrow}((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$. We only need to prove $\mathbf{f}^{\leftarrow}((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$

$$\begin{aligned} \mathbf{f}^{\leftarrow}((\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})) &= \mathbf{f}^{\leftarrow}((\bigsqcup_{k=0}^{\infty}(\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e})) \\ &= (\mathbf{f}^{\leftarrow} \circ (\bigsqcup_{k=0}^{\infty}(\mathbf{f}^{\rightarrow})^{(k)}))(\mathbf{e}) \qquad (4.2.12) \\ &= (\bigsqcup_{k=0}^{\infty}(\mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)}))(\mathbf{e}) \text{ by Lemma 65} \end{aligned}$$

We prove by induction that $(\mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$.

Basis step. $k = 0$. $(\mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(0)})(\mathbf{e}) = \mathbf{e} \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$.

Induction step. Our induction hypothesis (IH) is $(\mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$.

$$\begin{aligned}
(\mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k+1)})(\mathbf{e}) &= (\mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)} \sqcup (\mathbf{f}^{\rightarrow})^{(k)}))(\mathbf{e}) \\
&= (\mathbf{f}^{\leftarrow} \circ \mathbf{f}^{\rightarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)} \sqcup \mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \text{ by Lemma 11} \\
&\subset ((\mathbf{f}^{\rightarrow})^{*} \circ (\mathbf{f}^{\rightarrow})^{(k)} \sqcup \mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \text{ by Lemma 62} \\
&\subset ((\mathbf{f}^{\rightarrow})^{*} \circ (\mathbf{f}^{\rightarrow})^{*} \sqcup \mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \\
&\subset ((\mathbf{f}^{\rightarrow})^{*} \sqcup \mathbf{f}^{\leftarrow} \circ (\mathbf{f}^{\rightarrow})^{(k)})(\mathbf{e}) \text{ by Lemma 64} \\
&\subset ((\mathbf{f}^{\rightarrow})^{*} \sqcup (\mathbf{f}^{\rightarrow})^{*})(\mathbf{e}) \text{ by (IH)} \\
&= (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})
\end{aligned}$$

□

Corollary 68. $(\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$ *is a fixpoint of* (4.1.5).

Proof. This corollary follows directly from Lemma 66 and Lemma 67. □

Theorem 69. *If* $\mathbf{f}$ *is the coefficient of a set equations of the form* (4.1.5), *then* $\mathbf{f}^{*}(\mathbf{e}) = (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$.

Proof. By Corollary 68, $(\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$ is a fixpoint of $\mathbf{f}$. By monotonicity, $(\mathbf{f}^{\rightarrow})^{*}(\mathbf{e}) \subset \mathbf{f}^{*}(\mathbf{e})$. Since $\mathbf{f}^{*}(\mathbf{e})$ is the least fixpoint of $\mathbf{f}$, $\mathbf{f}^{*}(\mathbf{e}) \subset (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$. Therefore, $\mathbf{f}^{*}(\mathbf{e}) = (\mathbf{f}^{\rightarrow})^{*}(\mathbf{e})$. □

4.3 The Generation of Terms

Given the function vector $\mathbf{f}^{\rightarrow}$, we convert it to a formal (not necessarily context-free) grammar so that we can generate terms inductively.

For every variable X in $\mathbf{X}$,

- Let p be the index of X in $\mathbf{X}$
- $\pi_p(\mathbf{f}^{\rightarrow})$ must have the form $\bigsqcup_{i=1}^{n} \overrightarrow{t_i}$ for some integer n and terms $t_1, \ldots, t_n$
- Generate formal grammar rule $X ::= t_1 \mid \ldots \mid t_n$

Example 70. If we have

1. $\mathbf{X} = [X, Y]$ and
2. $S = \{\{p(X), \neg p(f(X))\}, \{\neg p(X)\}, \{p(f(f(a)), q(Y)\}, \{\neg q(g(a, a))\}\}$,

then we have

$$\mathbf{f}^{\rightarrow} \quad = \quad [\overrightarrow{a} \sqcup \overrightarrow{f(a)} \sqcup \overrightarrow{f(X)}, \overrightarrow{g(a,a)}]$$

We can convert this to a formal grammar

$$\begin{aligned} X &::= a \mid f(a) \mid f(Y) \\ Y &::= g(a, a) \end{aligned}$$

This allows us restrict the instances of X to the sequence of terms $a, f(a), f(f(a)), \ldots$ and those of Y to the term $g(a, a)$.

The terms generated this way are used in our d function from Section 3.4.1 to instantiate remaining free variables in an input clause after unifying some of its literals with literals in the current model. Using $\mathbf{f}^{\rightarrow}$ in d allows the prover to reduce the search space by avoiding instantiating free variables to terms that are not generated by $\mathbf{f}^{\rightarrow}$.

Example 71. Following Example 70, if the initial interpretation makes all negative literals true, then we know that the first contradicting instance must be an instance of $\{p(f(f(a)), q(Y)\}$. But how do we instantiate Y? In genOSHL, we will probably try the following sequence $a, f(a), f(f(a)), g(a, a)$. But in typedGenOSHL, we can only generate $g(a, a)$ for Y. This reduces the search space and improves the performance.

The remaining question is whether this restriction preserves the completeness of the genOSHL. If we reexamine the proof of completeness for genOSHL, we can notice that the key question here is whether or not we can still find a contradicting set of instances of the input clauses if we restrain variable instantiation to $\mathbf{f}^{\rightarrow}$. We will show below that the answer is affirmative.

The intuition of this proof is as follows: given an unsatisfiable set of input clauses, we can always find a resolution proof for it. To show that there is a set of contradicting ground instances, we try to construct a ground resolution proof from the resolution proof, restricting variable instantiation to terms generated by $\mathbf{f}$. We can do this because we have proved in the previous subsection that the terms generated by $\mathbf{f}^{\rightarrow}$ are the same as those generated by $\mathbf{f}$. We then prove by induction on the resolution proof that this construction can be done for any resolution proof.

To make our discussion easier, we give the definition of a resolution proof as follows:

Definition 72. Given a variable disjoint set S of input clauses, a resolution proof of S is a full binary tree where every node has a $(n+1)$-tuple $\langle C, L_1, \ldots, L_n\rangle$ of a clause and n literals such that

1. For the root, $C = \emptyset$ and $n = 0$;
2. For other nodes, $n > 0$ and $L_1, \ldots, L_n \in C$;
3. For leaves, C is a variant of some clause in S with fresh variables;

4. For any internal node $\langle D, M_1, M_2, \ldots, M_n\rangle$ with children $\langle C_1, L_{11}, L_{12}, \ldots L_{1k_1}\rangle$ and $\langle C_2, L_{21}, L_{22}, \ldots, L_{2k_2}\rangle$, where n, k_1, and k_2 are integers, there exists

$$\sigma \;=\; mgu_{reg}(L_{11}, \ldots, L_{1k_1}, \overline{L_{21}}, \ldots, \overline{L_{2k_2}})$$

such that for any other literals L in C_1, $\sigma(L) \neq \sigma(L_{11})$, for any other literal L in C_2, $\sigma(L) \neq \sigma(L_{21})$, and

$$D \;=\; (\sigma(C_1)\backslash\{\sigma(L_1)\}) \cup (\sigma(C_2)\backslash\{\sigma(L_2)\})$$

The third condition guarantees that clauses in leaves are variant disjoint. For each occurrence of literal L in a resolution proof, we define its originating literal set L^0 as follows:

Definition 73. Given a a resolution proof,

1. For leaves $\langle D, M_1, M_2, \ldots, M_n\rangle$, the origination literals $M_i^0 = M_i$ for $1 \leq i \leq n$.

2. For any internal node $\langle D, M_1, M_2, \ldots, M_n\rangle$ with children $\langle C_1, L_{11}, L_{12}, \ldots L_{1k_1}\rangle$ and $\langle C_2, L_{21}, L_{22}, \ldots, L_{2k_2}\rangle$, for $1 \leq i \leq n$,

$$M_i^0 \;=\; \bigcup_{L \in C_1 \cup C_2, M_i = \sigma(L)} L^0$$

where

$$\sigma \;=\; mgu_{reg}(L_{11}, \ldots, L_{1k_1}, \overline{L_{21}}, \ldots, \overline{L_{2k_2}})$$

It is possible that L^0 contains multiple literals for some literal L which appears in internal nodes. For example, if we have leaf nodes $\langle\{p(X), q(X)\}, q(X)\rangle$ and

$\langle\{p(Y), \neg q(Y)\}, \neg q(Y)\rangle$ and internal node $\langle\{p(X)\}, \ldots\rangle$ such that the leaf nodes are subtrees of the internal node, then, for the occurrence of $p(X)$ in the internal node, $(p(X))^0 = \{p(X), p(Y)\}$.

We give the definition of ground resolution proofs as follows:

Definition 74. Given a input clause S, a ground resolution proof of S is a full binary tree where every node a pair $\langle C, L\rangle$ of a clause and a literal such that

1. For the root, $C = \emptyset$ and $L = \perp$;
2. For other nodes, $L \in C$;
3. For leaves, C is a ground instance of some clause in S;
4. For an internal node $\langle D, M\rangle$ with children $\langle C_1, L_1\rangle$ and $\langle C_2, L_2\rangle$,

$$L_1 = \overline{L_2}$$

and

$$D = (C_1 \backslash \{L_1\}) \cup (C_2 \backslash \{L_2\})$$

Given a resolution proof, we want to find a way to instantiate variables in all leaves such that the instantiation uses terms in $\mathbf{f}^*(\mathbf{e})$ only. We define the type of a variable in a resolution proof as follows:

Definition 75. Given a variable disjoint set S of input clauses and a resolution proof of S, for each leaf $\langle C, L\rangle$, there is a clause $D \in S$ such that $C = \rho(D)$ for some renaming ρ. For each variable $X \in \mathbf{v}(C)$, we call the inverse image of X under ρ the type of X. We denote the type of a variable X by τ_X.

We start our construction of a ground resolution proof from a resolution proof by generating equational constraints from originating literal sets as follows:

For every internal node $\langle D, M_1, M_2, \ldots, M_n\rangle$ with children $\langle C_1, L_{11}, L_{12}, \ldots, L_{1k_1}\rangle$ and $\langle C_2, L_{21}, L_{22}, \ldots, L_{2k_2}\rangle$,

1. Generate regular most general unifier as follows:

   (a) Let $L_1^0 = \bigcup_{1\leq i\leq k_1} L_{1i}^0$. Let L_1 be an arbitrary literal in L_1^0.

   (b) Let $L_2^0 = \bigcup_{1\leq i\leq k_2} L_{2i}^0$. Let L_2 be an arbitrary literal in L_2^0.

   (c) For all $L_2' \in L_2^0$, generate $mgu_{reg}(L_1, \overline{L_2'})$

   (d) For all $L_1' \in L_1^0$, generate $mgu_{reg}(L_1', \overline{L_2})$

2. For every regular most general unifier σ generated, for every variable $X \in \mathbf{X}$ such that $\sigma(X) = s$ and $s \neq X$, generate equational constraint $X = s$.

This construction generates a list of equational constraints. Any solution σ to these equations will guarantee that $\sigma(L_1) = \sigma(\overline{L_2'})$ for all $L_2' \in L_2^0$ and $\sigma(L_1') = \sigma(\overline{L_2})$ for all $L_1' \in L_1^0$. By transitivity, $\sigma(L_1') = \sigma(\overline{L_2'})$ for all $L_2' \in L_2^0$ and all $L_1' \in L_1^0$. These set of constraints are also minimal in the following sense: If $\sigma(L_1') = \sigma(\overline{L_2'})$ for all $L_2' \in L_2^0$ and all $L_1' \in L_1^0$, then σ must also be a solution to this set of constraints.

Lemma 76. *Any ground instantiation of the resolution proof must satisfy these constraints.*

Proof. We can prove this easily by contradiction. □

Lemma 77. *If constraints $X = t_1, \ldots, X = t_n$ are generated, then $t_1, \ldots, t_n$ are unifiable.*

Proof. Otherwise there is no instantiation that can satisfy these constraints. □

This means that we can simplify the constraints so that each variable appears at most once on the left-hand side of an equation. The following lemma shows that we cannot have any non-trivial cycles in our constraints:

Lemma 78. *If constraints $X_1 = t_1, \ldots, X_n = t_n$ are generated and X_1 appears in t_2, X_2 appears in t_3, ..., and X_n appears in t_1, then $t_1 = X_2$, $t_2 = X_3$, ..., and $t_n = X_1$.*

Proof. Otherwise there is no instantiation that can satisfy these constraints. $\square$

Constraints that form cycles only have one form: equality between two variables. This means that we sort variables by the following order.

Definition 79. Given a set of equational constraints generated from a resolution proof, define order $\prec$ on variables as follows: $X \prec Y$ if and only if Y is bound to a non-variable term t containing X.

Given this order, we can generate an instantiation of the variables starting from the minimum variable and build up to the maximum variable. The remaining question is whether this solution is in $\mathbf{f}^*(\mathbf{e})$.

Lemma 80. *If $\overrightarrow{t}$ appears in $\pi_p(\mathbf{f}^*)$ for some index p, then $\pi_q \circ \overleftarrow{t} \circ \pi_p$* appears in $\pi_q(\mathbf{f}^*)$ for every index q such that $\pi_q(\mathbf{X}) \in \mathbf{v}(t)$.

Proof. By construction **F**. $\square$

Lemma 81. *If $\overrightarrow{X}$ appears in $\pi_q(\mathbf{f}^*)$, $X = \pi_p(\mathbf{X})$, and $Y = \pi_q(\mathbf{X})$ for indices q and p, then $\{\overrightarrow{X}, \overrightarrow{Y}\} \subset \pi_q(\mathbf{f}^*) = \pi_p(\mathbf{f}^*)$.*

Proof. By construction **E**. $\square$

The following lemma is based on the following idea: Suppose that we fix a variable X and take a subset of the constraints of the form $X = t$ or $Y = X$. If we know that $\overrightarrow{t_1}, \ldots, \overrightarrow{t_n}$ are all term functions that appear in $\pi_p(\mathbf{f})$ where p is the index of X in $\mathbf{X}$, then we can simplify these constraints to another set of constraints in which X does not appear.

Lemma 82. *Given a list $X_1 = t_1, \ldots, X_n = t_n$ of constraints and indices $p_1, \ldots, p_i$ such that*

1. *$\pi_{p_i}(\mathbf{X}) = X_i$ for $1 \leq i \leq n$ and*
2. *$\overrightarrow{t_i}$ appears in $\pi_{p_i}(\mathbf{f})$ for $1 \leq i \leq n$,*

we can find a solution $[\mathbf{t}/\mathbf{X}]$ to these constraints such that $\pi_i(\mathbf{t})$ appears in $\mathbf{f}^*$, for $1 \leq i \leq dim(\mathbf{X})$.

Proof. Prove by induction on the number of distinct variables.

Basis step. $n = 0$. Trivial.

Induction step. Our induction hypothesis (IH) is if there are less than k distinct variables, then this lemma holds. We need to prove that it holds for k variables.

First, delete all trivial constraints of the form $X = X$. We pick a variable X in $\mathbf{X}$ which does not appear in any term except itself. Let the index of X be p. Given two constraints involving X, there are three cases:

Case 1. $X = s, X = t$ for some terms s and t. Unify s with t, generating regular most general unifier σ. For every variable Y such that $\sigma(Y) \neq Y$, generate new constraint $Y = \sigma(Y)$. Let the index of Y be q. We need to show that $\sigma(Y)$ appears in $\pi_q(\mathbf{f})$. By premise 2, we know that $\overrightarrow{s}$ and $\overrightarrow{t}$ appear in $\pi_p(\mathbf{f})$. Since $\sigma(Y) \neq Y$, either $Y \in \mathbf{v}(s)$ or $Y \in \mathbf{v}(t)$. Without loss of generality, suppose $Y \in \mathbf{v}(s)$. By Lemma 80, $\pi_q \circ \overleftarrow{s} \circ \pi_p$ appears in $\pi_q(\mathbf{f})$. By construction $\mathbf{G}$, $\sigma(Y)$ appears in $\pi_q(\mathbf{f})$.

Case 2. $X = s, Y = X$ for some term s and some variable Y. Let the index of Y be q. Generate new constraint $Y = s$. By premise 2, we know that $\overrightarrow{s}$ appears in $\pi_p(\mathbf{f})$ and $\overrightarrow{X}$ appears in $\pi_q(\mathbf{f})$. By Lemma 81, $\overrightarrow{s}$ appears in $\pi_q(\mathbf{f})$.

Case 3. $Y = X, Z = X$ for some variables Y and Z. Let the index of Y be q. Let the index of Z be r. Generate new constraint $Y = Z$. By premise 2, we know that $\overrightarrow{X}$ appears in $\pi_q(\mathbf{f})$ and $\pi_r(\mathbf{f})$. By Lemma 81, $\overrightarrow{Z}$ appears in $\pi_q(\mathbf{f})$.

We fix one constraint involving X and perform the foregoing operation on this constraint and every other constraint involving X. The newly generated constraints are sufficient to guarantee that constraints involving X are satisfied. We remove all constraints involving X.

Now we have a new list of constraints that has only k variables. By (IH), we can find a solution $[\mathbf{t}/\mathbf{X}]$, where and all terms in $\mathbf{t}$ appears in $\mathbf{f}^*$.

To construct a solution for the original set of constraints, we arbitrarily pick a constraint involving X in the original set of constraints.

Case 1. The constraint involving X is $X = s$. We can find a solution to the original set of constraints by substituting $\overrightarrow{s}(\mathbf{t})$ for X: $[\mathbf{t}/\mathbf{X}][\overrightarrow{s}(\mathbf{t})/p]$. By premise 2, $\overrightarrow{s}$ appears in $\mathbf{f}$. Therefore, $\overrightarrow{s}(\mathbf{t})$ appears in $\mathbf{f}^*$.

Case 2. The constraint is $Y = X$. Let the index of Y be q. We can find a solution to the original set of constraints by substituting $\overrightarrow{Y}(\mathbf{t})$ for X: $[\mathbf{t}/\mathbf{X}][\overrightarrow{Y}(\mathbf{t})/p]$. By premise 2, $\overrightarrow{X}$ appears in $\pi_q(\mathbf{f})$. By Lemma 81, $\overrightarrow{Y}$ appears in $\pi_p(\mathbf{f})$. Therefore, $\overrightarrow{Y}(\mathbf{t})$ appears in $\mathbf{f}^*$. □

Lemma 83. *We can find an instantiation of a resolution proof to a ground resolution proof even if we restrict instantiation to* $\mathbf{f}^*(\mathbf{e})$.

Proof. By Lemma 82 we can find a set of contradicting instances of clauses in S. We arbitrarily instantiate remaining variables by terms generated by their type which results in a set T of contradicting ground instances of clauses in S. □

Theorem 84. *typedGenOSHL is complete.*

Proof. Let S' be the set of all ground instances generated from S by restricting instantiation to $\mathbf{f}^*(\mathbf{e})$. By Lemma 83, we can find a contradicting set T of ground instances which is a subset of S'. If we use this S' in our function d, then type-dGenOSHL is complete. $\square$

4.4 Algorithm $\mathcal{G}$

Ideally, the construction $\mathbf{G}$ in Definition 55 would terminate when no new term functions can be added any more after finite number of iterations. One problem with most general unifiers is that they can map a variable to a term that has size exponential to the size of the two input terms. A classic example [36] is the following: Consider terms

$$h(x_1, \ldots, x_n, f(y_0, y_0), \ldots, f(y_{n-1}, y_{n-1}), y_n)$$

and

$$h(f(x_0, x_0), \ldots, f(x_{n-1}, x_{n-1}), y_1, \ldots, y_n, x_n)$$

Their most general unifier maps each x_i or y_i to a term of size $2^{i+1} - 1$. This means that it is possible that our construction may keep generating new, larger term functions, which may lead to a sequence $\mathbf{G}_k(P)$ that contains infinitely many growing sets. Consequently, our construction of $\mathbf{G}(P)$ may not terminate in finite steps. To alleviate this problem, we introduce a different kind of unifier: θ-mgu.

This section introduces θ-mgu and shows that if we replace the most general unifiers in our construction $\mathbf{G}$ from Definition 55 by θ-mgus, then the minimum solution to equation 4.1.5 is at least as large as the minimum solution when we do not replace most general unifiers by θ-mgus; and if we replace the most general unifiers in our construction $\mathbf{G}$ from Definition 55 by θ-mgus, then the construction always terminates. Readers not interested in the termination property can skip this section.

4.4 The Algorithm

Algorithm $\mathcal{G}$ is the algorithm we use to compute θ-mgus of a set of pairs of terms. It is tailored to the specific need of our type inference preprocessor. We follow the following conventions in our description of the algorithm $\mathcal{G}$:

1. Assume that every function symbol (or predicate symbol) has a fixed arity.
2. Given a substitution θ, a term t, and a variable X such that $\theta(X) = X$ and $X \neq t$, denote the substitution obtained by appending t/X to θ by $\theta[t/X]$. For example, if $\theta = [f(X)/Y]$, then $\theta[f(Y)/X] = [f(X)/Y, f(Y)/X]$.
3. Denote failure by $\perp$.

Definition 85. We define the algorithm $\mathcal{G}$ as a derivation $\Rightarrow$. It starts with a triple $G, [], []$, where G is a set of pairs of terms and $[]$ is an empty substitution, and each derivation step falls into one of the following eight cases.

1. $G \cup \{\langle t, t\rangle\}, \theta, \sigma \Rightarrow G, \theta, \sigma$
2. $G \cup \{\langle f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)\rangle\}, \theta, \sigma \Rightarrow G \cup \{\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle\}, \theta, \sigma$
3. $G \cup \{\langle f(s_1, \ldots, s_m), g(t_1, \ldots, t_n)\rangle\}, \theta, \sigma \Rightarrow \emptyset, \perp, \perp$, where $f \neq g$
4. $G \cup \{\langle f(s_1, \ldots, s_n), X\rangle, \theta, \sigma\} \Rightarrow G \cup \{\langle X, f(s_1, \ldots, s_n)\rangle, \theta, \sigma\}$, where $\theta(X) = X$
5. $G \cup \{\langle X, t\rangle\}, \theta, \sigma \Rightarrow G \cup \{\langle \theta(X), t\rangle\}, \theta, \sigma$, where $\theta(X) \neq X$
6. $G \cup \{\langle t, Y\rangle\}, \theta, \sigma \Rightarrow G \cup \{\langle X, \theta(Y)\rangle\}, \theta, \sigma$, where t is either a non-variable term or a variable such that $\theta(t) = t$, and $\theta(Y) \neq Y$
7. $G \cup \{\langle X, t\rangle\}, \theta, \sigma \Rightarrow G, \theta[t/X], [\sigma(t)/X] \circ \sigma$, where t is either a non-variable term or $\theta(t) = t$, $\theta(X) = X$, and $X \notin \mathbf{v}(\sigma(t))$

8. $G \cup \{\langle X, t\rangle\}, \theta, \sigma \Rightarrow \emptyset, \bot, \bot$, where t is not a variable, $\theta(X) = X$, and $X \in \mathbf{v}(\sigma(t))$

The first case is when we try to unify one term to itself. This is a trivial case. We just need to delete the pair of the term and itself from G. The second case is when we try to unify two terms with the same top-level function symbols or predicate symbols. Since we assume that every function symbol (or predicate symbol) has a fixed arity, we do not have to deal with the case where there are different numbers of subterms. The third case is when we try to unify two terms with different top-level function symbols or predicate symbols. This leads to failure. The fourth case switches a non-variable term with an unbound variable. The fifth case is when we try to unify a bound variable X with a term t. We replace the pair with a new pair, replacing the variable X with the term $\theta(X)$. The sixth case is when we try to unify a non-variable term t with a bound variable Y. We replace the pair with a new pair, replacing the variable Y with the term $\theta(Y)$. The seventh case is when we try to unify an unbound variable X with a non-variable term t, where X does not occur in t. In this case, we update unifiers θ and σ to map the variable to the term. The eighth case is when we try to unify an unbound variable X with a non-variable term t, where X occurs in t. This leads to failure.

Following the standard notational conventions, we denote k step derivation, where $k \geq 0$ is an integer, from G, θ, σ to G', θ', σ' by $G, \theta, \sigma \Rightarrow^k G', \theta', \sigma'$ and zero or more step derivation from G, θ, σ to G', θ', σ' by $G, \theta, \sigma \Rightarrow^* G', \theta', \sigma'$. The derivation terminates when either $\emptyset, \bot, \bot$ or $\emptyset, \theta, \sigma$ for some substitutions θ and σ is reached.

Algorithm $\mathcal{G}$ differs from the standard unification algorithm in the following manner. No substitution is applied to the remaining pairs of terms to be unified when the unifier being constructed is updated. Also, Algorithm $\mathcal{G}$ computes two unifiers θ and σ.

σ is a most general simultaneous unifier of all pairs in the initial set G. On the one hand, note that any variable X bound by θ is replaced by $\theta(X)$. This is equivalent to repeatedly applying θ to G. Our main theorem will show that the effect of this is equivalent to applying σ to G. On the other hand, it can be easily proved that $\theta(X) = X$ if and only if $\sigma(X) = X$. Therefore, we can replace G with $\sigma(G)$ and tests like $\theta(X) = X$ with $\sigma(X) = X$ and obtain a variant of the standard unification algorithm. Also, σ is regular (Definition 48). In contrast, in general, θ is not necessarily a unifier of s and t and is not idempotent. However, the main theorem of this section will show that θ and σ are equivalent in a certain sense, hence θ has as much information as σ does. θ has a property which σ does not have: if we replace the most general unifier in construction **G** (Definition 55), then **G** will always generate a smaller term function, from which the termination of **G** easily follows.

In order to distinguish these two different kinds of most general unifiers, we define the following:

Definition 86. Given terms s and t such that $\{\langle s,t\rangle\}, [], [] \Rightarrow^* \emptyset, \theta, \sigma$ for some $\theta \neq \bot$ and $\sigma \neq \bot$, we say that θ is the θ-mgu of s and t, written $mgu_\theta(s,t)$, and σ is the σ-mgu of s and t, written $mgu_\sigma(s,t)$.

4.4 Termination of Algorithm $\mathcal{G}$

In this subsection, we show that $\mathcal{G}$ terminates on all inputs. We can find a very simple termination proof for this algorithm based on the termination proof of the standard unification algorithm.

Theorem 87. *Algorithm $\mathcal{G}$ terminates.*

Proof. We define a vector $\mathbf{z}$ as follows.

Let l be the number of distinct variables in G. Assign to each variable X an integer z_X by doing the following repeatedly.

1. If $X \notin \mathbf{v}(\theta(Y))$ for all $Y \neq X$, assign $z_X = l$
2. If $X \in \mathbf{v}(\theta(Y))$ for some $Y \neq X$, and for every variable $Y \neq X$ such that $X \in \theta(Y)$, z_Y is assigned, assign $z_X = min_{\leq}(\{z_Y \mid X \in \mathbf{v}(\theta(Y))\}) - 1$.

According to this assignment, if $Y \in \mathbf{v}(\theta(X))$, then $z_Y < z_X$. This is useful because in Case 5 (and similarly Case 6) of Definition 85, we replace a variable X by $\theta(X)$. The property of our construction here ensures that for every variable Y in $\theta(X)$, we have $z_Y < z_X$. We define an l dimensional vector $\mathbf{z} = [z_i]_{i=1}^{l}$ as follows. z_i is the number of variables X (count repeated variables) in G such that $z_X = i$ for all indices i. We define an order $<_{\mathbf{z}}$ on all possible values of $\mathbf{z}$ as follows. $\mathbf{z} <_{\mathbf{z}} \mathbf{z}'$ if and only if there is an index k such that $\mathbf{z}_l = \mathbf{z}'_l, \mathbf{z}_{l-1} = \mathbf{z}'_{l-1}, \ldots, \mathbf{z}_{k+1} = \mathbf{z}'_{k+1}$ and $\mathbf{z}_k < \mathbf{z}'_k$. We say that an operation does not increase $\mathbf{z}$ if the operation does not make the resulting $\mathbf{z}$ larger with respect to $<_{\mathbf{z}}$.

We will show that all operations in G leads to one of the following

1. Decrease $\mathbf{z}$.
2. Keep $\mathbf{z}$ unchanged and decrease the total number of symbols in G.
3. Keep $\mathbf{z}$ and the total number of symbols in G unchanged and decrease the number of pairs whose first component is a variable.

We show this by case analysis on a single derivation step.

Case 1. Removing an element from $\mathcal{G}$ does not increase $\mathbf{z}$ and decreases the total number of symbols in G.

Case 2. Since $size(f(s_1, \ldots, s_n)) > size(s_1) + \ldots + size(s_n)$, and $size(g(t_1, \ldots, t_n)) > size(t_1) + \ldots + size(t_n)$, replacing $\langle f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)\rangle$ by $\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle$ keeps $\mathbf{z}$ unchanged and decreases the total number of symbols in G.

Case 3. Replacing a nonempty set with an empty set does not increase $\mathbf{z}$ and decreases the total number of symbols in G.

Case 4. Switching the two components keeps $\mathbf{z}$ and the total number of symbols in G unchanged and decreases the number of pairs who first component is a variable.

Case 5. Replacing $\langle X, t\rangle$ by $\langle \theta(X), t\rangle$ in $\mathcal{G}$ keeps $z_l, z_{l-1}, \dots z_{z_X+1}$ unchanged and decreases z_{z_X} since by construction, every variable Y in $\theta(X)$ has a smaller z_Y than z_X.

Case 6. Replacing $\langle X, Y\rangle$ by $\langle X, \theta(Y)\rangle$ in $\mathcal{G}$ keeps $z_l, z_{l-1}, \dots z_{z_Y+1}$ unchanged and decreases z_{z_Y} since by construction.

Case 7. Appending $[t/X]$ to θ does not increase the z_Y for any variable Y in t. Hence it does not increase $\mathbf{z}$. Removing an element from $\mathcal{G}$ does not increase $\mathbf{z}$ and decreases c.

Case 8. Removing an element from $\mathcal{G}$ does not increase $\mathbf{z}$ and decreases the total number of symbols in G. □

4.4 The Equivalence of σ and θ

The first lemma shows that θ and σ bind the same set of variables

Lemma 88. *If $G, [], [] \Rightarrow^* G', \theta, \sigma$ and $\theta \neq \bot$, then $\theta(X) = X$ if and only if $\sigma(X) = X$.*

Proof. We prove by induction on the derivation of $G, [], [] \Rightarrow^* G', \theta, \sigma$.

Basis step. $\theta = \sigma = []$. Trivial.

Induction step. Our induction hypothesis (IH) is that for every derivation $G, [], [] \Rightarrow^k G_k, \theta_k, \sigma_k$, $\theta_k(X) = X$ if and only if $\sigma_k(X) = X$. Now, consider the derivation step $G_k, \theta_k, \sigma_k \Rightarrow G_{k+1}, \theta_{k+1}, \sigma_{k+1}$. The only case where $\theta_{k+1} \neq \theta_k$ or $\sigma_{k+1} \neq \sigma_k$ is Case 7. Therefore, we only need to consider this case.

Case 7. Since $\theta_k(X) = X$, By (IH), $\sigma_k(X) = X$. By the condition that $X \notin \mathbf{v}(\sigma_k(t))$, $X \neq t$ and $X \neq \sigma_k(t)$. Therefore, $\theta_{k+1}(X) \neq X$ and $\sigma_{k+1}(X) \neq X$.

Suppose towards a contradiction that there is a variable Y such that $\sigma_k(Y) \neq Y$, but $\sigma_{k+1}(Y) = Y$. We have $([\sigma_k(t)/X] \circ \sigma_k)(Y) = Y$. Therefore,

$$\sigma_k(t) \quad = \quad Y \tag{4.4.1}$$

and

$$\sigma_k(Y) \quad = \quad X \tag{4.4.2}$$

By the condition that t is either a non-variable term or $\sigma_k(t) = t$ and (4.4.1),

$$t \quad = \quad Y \tag{4.4.3}$$

By (4.4.3) and (4.4.1)

$$\sigma_k(Y) \quad = \quad t \tag{4.4.4}$$

By (4.4.2) and (4.4.4), $X = t$, which is contradictory to $X \notin \mathbf{v}(\sigma_k(t))$. $\square$

The next lemma shows that σ is idempotent, i.e., $\sigma \circ \sigma = \sigma$.

Lemma 89. *If $G, [], [] \Rightarrow^* G', \theta, \sigma$ and $\theta \neq \bot$, then σ is idempotent.*

Proof. We prove by induction on the derivation of $G, [], [] \Rightarrow^* \emptyset, \theta, \sigma$.

Basis step. $\sigma = []$. It is idempotent.

Induction step. Our induction hypothesis (IH) is that for every derivation $G, [], [] \Rightarrow^k G_k, \theta_k, \sigma_k$, σ_k is idempotent. Now, consider the derivation step G_k, θ_k, σ_k

$\Rightarrow G_{k+1}, \theta_{k+1}, \sigma_{k+1}$. The only case where $\sigma_{k+1} \neq \sigma_k$ is Case 7. Therefore, we only need to consider this case.

Case 7. Since $\theta_k(X) = X$, by Lemma 88, $\sigma_k(X) = X$. Therefore,

$$\sigma_k \circ [\sigma_k(t)/X] \quad = \quad [\sigma_k(\sigma_k(t))/X] \circ \sigma_k \tag{4.4.5}$$

By (IH) and (4.4.5),

$$\sigma_k \circ [\sigma_k(t)/X] \quad = \quad [\sigma_k(t)/X] \circ \sigma_k \tag{4.4.6}$$

By our construction, X does not occur in $\sigma_k(t)$. Therefore,

$$[\sigma_k(t)/X] \circ [\sigma_k(t)/X] \quad = \quad [\sigma_k(t)/X] \tag{4.4.7}$$

Therefore

$$\begin{aligned}
\sigma_{k+1} \circ \sigma_{k+1} &= ([\sigma_k(t)/X] \circ \sigma_k) \circ ([\sigma_k(t)/X] \circ \sigma_k) \text{ by construction} \\
&= [\sigma_k(t)/X] \circ [\sigma_k(t)/X] \circ \sigma_k \circ \sigma_k \text{ by (4.4.6)} \\
&= [\sigma_k(t)/X] \circ \sigma_k \circ \sigma_k \text{ by (4.4.7)} \\
&= [\sigma_k(t)/X] \circ \sigma_k \text{ by (IH)} \\
&= \sigma_{k+1} \text{ by construction}
\end{aligned}$$

□

An immediate corollary of this lemma is that σ contains no loop.

Corollary 90. *If $G, [], [] \Rightarrow^* G', \theta, \sigma$ and $\theta \neq \perp$, then there is no loop with length greater than one in σ.*

Proof. Suppose towards a contradiction that there is a loop $\mathbf{p}_0, \ldots, \mathbf{p}_n$ of length $n > 1$. By Lemma (89), we can easily prove by induction that $\mathbf{p}_i \in \sigma(\mathbf{p}_1)$ for $1 \leq i < n$. Hence $rm(\mathbf{p}_i) \in \mathbf{v}(\sigma(rm(\mathbf{p}_1)))$. Since $rm(\mathbf{p}_n) = rm(\mathbf{p}_1)$, $rm(\mathbf{p}_1) \in \mathbf{v}(\sigma(rm(\mathbf{p}_1)))$. This is contradictory to the idempotency of σ. $\square$

The next lemma is our main lemma. It shows that θ contains no loop. Since θ is not always idempotent, we need to find a different way of proving its "looplessness."

Lemma 91. *If $G, [], [] \Rightarrow^* G', \theta, \sigma$ and $\theta \neq \bot$, then for every term s and every nonnegative integer r,*

1. *there is no loop with length r or less in θ and*
2. *for every path $\mathbf{p} \in pset(\theta^r(s))$ such that $rm(\mathbf{p})$ is a variable and $\theta(rm(\mathbf{p})) = rm(\mathbf{p})$, $\mathbf{p} \in pset(\sigma(s))$.*

Proof. We prove by a two level induction.

Level 1. We fix s and induct on r.

Basis step. $r = 0$. It is trivially true.

Induction step. Our induction hypothesis (IH1) is that the lemma holds for $r - 1$.

1. there is no loop with length n such that $0 \leq n \leq r - 1$ in θ and
2. for every path $\mathbf{p} \in pset(\theta^{r-1}(s))$ such that $rm(\mathbf{p})$ is a variable and $\theta(rm(\mathbf{p})) = rm(\mathbf{p})$, $\mathbf{p} \in pset(\sigma(s))$.

We need to prove that it also holds for r.

Level 2. We prove by induction on the derivation of $G, [], [] \Rightarrow^* G', \theta, \sigma$.

Basis step. $\theta = []$, $\sigma = []$. It is trivially true.

Induction step. Our induction hypothesis (IH2) is that for every derivation $G, [], [] \Rightarrow^k G_k, \theta_k, \sigma_k$,

1. there is no loop of length r in θ_k

2. for every path $\mathbf{p} \in pset(\theta_k^r(s))$ such that $rm(\mathbf{p})$ is a variable and $\theta_k(rm(\mathbf{p})) = rm(\mathbf{p})$, $\mathbf{p} \in pset(\sigma_k(s))$.

Now, consider the derivation step $G_k, \theta_k, \sigma_k \Rightarrow G_{k+1}, \theta_{k+1}, \sigma_{k+1}$. The only case where $\theta_{k+1} \neq \theta_k$ or $\sigma_{k+1} \neq \sigma_k$ is Case 7. Therefore, we only need to consider this case.

Case 7. By construction, there is a variable X and a term t such that $\theta_{k+1} = [t/X] \circ \theta_k$.

1. We prove that there is no loop of length r in θ_{k+1}.

Suppose towards a contradiction that θ_{k+1} contains a loop $\mathbf{p}_0, \ldots, \mathbf{p}_n$. By (IH2), θ_k does not contain loops of length r. There must be an index $0 \leq i < r$ such that $\mathbf{p}_{i+1} \notin \theta_k(\mathbf{p}_i)$ but $\mathbf{p}_{i+1} \in \theta_{k+1}(\mathbf{p}_i)$. By the condition in our construction that

$$\theta_k(X) = X \tag{4.4.8}$$

the only difference between θ_k and $\theta_k[t/X]$ is that the former maps X to itself but the latter maps X to t. Therefore, it must be the case that

$$rm(\mathbf{p}_i) = X \tag{4.4.9}$$

$$rm(\mathbf{p}_{i+1}) \in \mathbf{v}(t) \tag{4.4.10}$$

We can always rotate the loop so that $i = 0$.

$$rm(\mathbf{p}_0) = X \tag{4.4.11}$$

$$rm(\mathbf{p}_1) \in \mathbf{v}(t) \tag{4.4.12}$$

Without loss of generality, we may assume

$$\mathbf{p}_0 \quad = \quad X \tag{4.4.13}$$

and $\mathbf{p}_1 \in pset(t)$. Now, consider other paths in the loop. According to the definition of a loop, $rm(\mathbf{p}_1), \ldots, rm(\mathbf{p}_{r-1})$ are different from $rm(\mathbf{p}_0)$ which is X. Therefore, they are not affected by the change from θ_k to θ_{k+1}, i.e.,

$$\theta_k(\mathbf{p}_j) \quad = \quad \theta_{k+1}(\mathbf{p}_j) \text{ for all } 0 < j < r \tag{4.4.14}$$

Therefore,

$$\mathbf{p}_{j+1} \quad = \quad \theta_k(\mathbf{p}_j) \text{ for all } 0 < j < r \tag{4.4.15}$$

Therefore,

$$\begin{aligned} \mathbf{p}_r &= \theta_k^{r-1}(\mathbf{p}_1) \\ &\in \theta_k^{r-1}(pset(t)) \text{ by } (4.4.12) \end{aligned} \tag{4.4.16}$$

Since $rm(\mathbf{p}_r) = rm(\mathbf{p}_0) = X$,

$$\theta_k(\mathbf{p}_r) \quad = \quad \{\mathbf{p}_r\} \tag{4.4.17}$$

Therefore,

$$\begin{aligned}
\mathbf{p}_r &\in \theta_k(\mathbf{p}_r) \text{ by (4.4.17)} \\
&\subset \theta_k^r(pset(t)) \text{ by (4.4.16)} \\
&= pset(\theta_k^r(t)) \text{ by Lemma 17} \\
&= pset(\sigma_k(t)) \text{ by (IH2)}
\end{aligned}$$

Therefore, $X \in \mathbf{v}(\sigma_k(t))$. This is contradictory to the condition in our construction that not $X \notin \mathbf{v}(\sigma_k(t))$.

2. We prove that for every path $\mathbf{p} \in pset(\theta_{k+1}^r(s))$ such that $rm(\mathbf{p})$ is a variable and $\theta_{k+1}(rm(\mathbf{p})) = rm(\mathbf{p})$, $\mathbf{p} \in pset(\sigma_{k+1}(s))$.

Suppose that $\mathbf{p}$ is a path such that

$$\begin{aligned}
\mathbf{p} &\in pset(\theta_{k+1}^r(s)) && (4.4.18) \\
\theta_{k+1}(rm(\mathbf{p})) &= rm(\mathbf{p}) && (4.4.19)
\end{aligned}$$

We need to show that $\mathbf{p} \in pset(\sigma_{k+1}(s))$. Let the history of $\mathbf{p}$ with respect to a list of r θ_{k+1}s and s be $\mathbf{p}_0, \ldots, \mathbf{p}_r$. Since $rm(\mathbf{p}_r)$ is a variable, $rm(\mathbf{p}_i)$ must also be a variable for $0 \leq i < r$. There must be an integer $r' \leq r$ such that

$$\begin{aligned}
\mathbf{p}_i &\neq \mathbf{p}_j \text{ for } 0 \leq i < j \leq r' && (4.4.20) \\
\mathbf{p}_i &= \mathbf{p}_j \text{ for } r' < i < j \leq r && (4.4.21)
\end{aligned}$$

We have proved that there is no loop of length r in θ_{k+1}. Therefore, $rm(\mathbf{p}_0), \ldots,$ $rm(\mathbf{p}_{r'})$ must be mutually distinct.

Case 7.1. $rm(\mathbf{p}_i) \neq X$ for $0 \leq i \leq r$. It can be easily proved by induction that there is a history of $\mathbf{p}$ with respect to a list of r θ_ks and t which coincides with

$\mathbf{p}_0, \ldots, \mathbf{p}_r$:

$$\mathbf{p}_i \in pset(\theta_k^i(s)) \text{ for } 0 \leq i \leq r \tag{4.4.22}$$

In particular,

$$\mathbf{p}_r \in pset(\theta_k^r(s)) \tag{4.4.23}$$

Since $rm(\mathbf{p}_r) \neq X$,

$$\begin{aligned} \theta_k(rm(\mathbf{p}_r)) &= \theta_{k+1}(rm(\mathbf{p}_r)) \\ &= rm(\mathbf{p}_r) \text{ by (4.4.19)} \end{aligned} \tag{4.4.24}$$

By (4.4.23), (4.4.24), and (IH2),

$$\mathbf{p}_r \in pset(\sigma_k(s)) \tag{4.4.25}$$

Again, since $rm(\mathbf{p}_r) \neq X$,

$$[\sigma_k(t)/X](\mathbf{p}_r) = \{\mathbf{p}_r\} \tag{4.4.26}$$

Therefore,

$$\begin{aligned} \mathbf{p}_r &\in [\sigma_k(t)/X](pset(\sigma_k(s))) \text{ by (4.4.25) and (4.4.26)} \\ &= pset([\sigma_k(t)/X](\sigma_k(s))) \text{ by Lemma 17} \\ &= pset(([\sigma_k(t)/X] \circ \sigma_k)(s)) \text{ by (2.5.1)} \\ &= pset(\sigma_{k+1}(s)) \text{ by construction} \end{aligned}$$

Case 7.2. There exists an integer i such that

$$rm(\mathbf{p}_i) \quad = \quad X \tag{4.4.27}$$

By the condition of our construction,

$$X \quad \notin \quad \mathbf{v}(\theta_{k+1}(X)) \tag{4.4.28}$$

By (4.4.19) and (4.4.28),

$$X \quad \neq \quad rm(\mathbf{p}_{r'}) \tag{4.4.29}$$

Therefore, i is unique.

Now, let us take a look at $\mathbf{p}_j$ for $0 \leq j \leq i$. Since $rm(\mathbf{p}_j) \neq X$ for $0 \leq j \leq i$,

$$\mathbf{p}_j \quad \in \quad \theta_k^j(s) \text{ for } 0 \leq j \leq i \tag{4.4.30}$$

In order to apply (IH1), observe that

$$\begin{aligned} \theta_k(rm(\mathbf{p}_i)) \quad &= \quad \theta_k(X) \text{ by (4.4.27)} \\ &= \quad X \text{ by the condition in our construction} \\ &= \quad rm(\mathbf{p}_i) \text{ by (4.4.27)} \end{aligned} \tag{4.4.31}$$

Now, by (4.4.30), (4.4.31), and the second clause of (IH1),

$$\mathbf{p}_i \quad \in \quad pset(\sigma_k(s)) \tag{4.4.32}$$

Next, let us take a look at $\mathbf{p}_j$ for $i < j \le r'$. Since $rm(\mathbf{p}_j) \neq X$ for $i < j \le r'$,

$$\mathbf{p}_j \in \theta_k^{j-(i+1)}(\mathbf{p}_{i+1}) \text{ for } i < j \le r \tag{4.4.33}$$

In order to apply (IH1), observe that

$$\begin{aligned} \theta_k(rm(\mathbf{p}_{r'})) &= \theta_{k+1}(rm(\mathbf{p}_{r'})) \text{ by (4.4.33)} \\ &= rm(\mathbf{p}_{r'}) \text{ by (4.4.19)} \end{aligned} \tag{4.4.34}$$

Now, by (4.4.33), (4.4.34), and the second clause of (IH1),

$$\mathbf{p}_{r'} \in \sigma_k(\mathbf{p}_{i+1}) \tag{4.4.35}$$

Also,

$$\begin{aligned} \mathbf{p}_{i+1} &\in \theta_k[t/X](\mathbf{p}_i) \\ &= [t/X](\mathbf{p}_i) \text{ by (4.4.27)} \end{aligned} \tag{4.4.36}$$

Combining all equations, we have

$$\begin{aligned}
\mathbf{p}_r &= \mathbf{p}_{r'} \\
&\in \sigma_k(\mathbf{p}_{i+1}) \text{ by (4.4.35)} \\
&\subset \sigma_k([t/X](\mathbf{p}_i)) \text{ by (4.4.36)} \\
&\subset \sigma_k([t/X](pset(\sigma_k(s)))) \text{ by (4.4.32)} \\
&= pset(\sigma_k([t/X](\sigma_k(s)))) \text{ by Lemma 17} \\
&= pset((\sigma_k \circ [t/X] \circ \sigma_k)(s)) \text{ by (2.5.1)} \\
&= pset(([\sigma_k(t)/X] \circ \sigma_k)(s)) \text{ by Lemma (89)} \\
&= pset(\sigma_{k+1}(s))
\end{aligned}$$

An immediate corollary following Lemma 91 is that applying θ repeatedly does not generate an infinite sequence of distinct terms □

Corollary 92. *If* $G, [], [] \Rightarrow^* G', \theta, \sigma$ *and* $\theta \neq \bot$*, there exists an integer* K *such that* $\theta^{K+1} = \theta^K$.

Proof. Arbitrarily pick a term t. Let K be the total number of distinct variables that appear in θ and t. Suppose towards a contradiction that $\theta^{K+1}(X) \neq \theta^K(X)$ for some variable X. There must be a path $\mathbf{p}$ such that $\mathbf{p} \in pset(\theta^{K+1}(X))$ but $\mathbf{p} \notin pset(\theta^K(X))$. Let the history of $\mathbf{p}$ with regard to $k+1$ θs and t be $\mathbf{p}_0, \ldots, \mathbf{p}_{K+1}$. Since $\mathbf{p} \notin pset(\theta^K(X))$, $\mathbf{p}_{K+1} \neq \mathbf{p}_K$. $rm(\mathbf{p}_0), \ldots, rm(\mathbf{p}_K)$ must all be variables. There are $k+1$ variables here, there must be two variables that are the same, which means that there is a loop in θ. This is contradictory to Lemma 91. □

Next, some equations that we will use in the proof of the main theorem of this subsection.

Lemma 93. *1. If σ is idempotent, then*

$$[\sigma(t)/X] \circ \sigma \quad = \quad \sigma \circ [t/X] \circ \sigma \tag{4.4.37}$$

2. If $\theta(X) = X$, then

$$\theta \circ [t/X] \quad = \quad [t/X] \circ \theta \tag{4.4.38}$$

$$\theta \circ [t/X] \quad = \quad \theta[t/X] \tag{4.4.39}$$

Proof. 1. Suppose that variable Y is in $\sigma(X)$. By the idempotency of σ, we have $\sigma(Y) = Y$.

Case 1. $Y = X$. $[\sigma(t)/X](Y) = \sigma(t) = \sigma([t/X](X)) = \sigma([t/X](Y))$.

Case 2. $Y \neq X$. $[\sigma(t)/X](Y) = Y = \sigma(Y) = \sigma([t/X](Y))$.

2. It is easy to see that these equations hold true since $\theta(X) = X$. □

θ is equivalent to σ in the following sense:

Theorem 94. *If $G, [], [] \Rightarrow^* G', \theta, \sigma$ and $\theta \neq \bot$, there exists an integer K such that $\theta^K = \sigma$.*

Proof. By induction on the derivation of $G, [], [] \Rightarrow^* \emptyset, \theta, \sigma$.

Basis step. $\theta = [] = \sigma$. $K = 0$.

Induction step. Our induction hypothesis (IH) is if $G, [], [] \Rightarrow^k G_k, \theta_k, \sigma_k$ and $\theta_k \neq \bot$, there exists an integer K such that $\theta_k^K(X) = \sigma_k(X)$ for all variables X in G. Now consider the derivation step $G_k, \theta_k, \sigma_k \Rightarrow G_{k+1}, \theta_{k+1}, \sigma_{k+1}$. Case 1, 2, 3, 4, 5, 6, 8 do not change θ_k or σ_k.

We only need to consider Case 7.

Case 7. By Corollary 92, there exists an integer K' such that

$$\theta_k^{K'+1} \;=\; \theta_k^{K'} \tag{4.4.40}$$

By the condition in our construction that

$$\theta_k(X) \;=\; X \tag{4.4.41}$$

$$\begin{aligned}
[\sigma_k(t)/X] \circ \sigma_k &= ([\sigma_k(t)/X] \circ \sigma_k)^{K'} \text{ by Lemma 89} \\
&= (\sigma_k \circ [t/X] \circ \sigma_k)^{K'} \text{ by (4.4.37) in Lemma 93} \\
&= (\theta_k^K \circ [t/X] \circ \theta_k^K)^{K'} \text{ by (IH)} \\
&= [t/X]^{K'} \circ \theta_k^{2KK'} \text{ by (4.4.41) and (4.4.38) in Lemma 93} \\
&= [t/X]^{K'} \circ \theta_k^{K'} \text{ by } 2KK' > K' \text{ and (4.4.40)} \\
&= ([t/X] \circ \theta_k)^{K'} \text{ by (4.4.41) and (4.4.38) in Lemma 93} \\
&= (\theta_k[t/X])^{K'} \text{ by (4.4.41) and (4.4.39) in Lemma 93}
\end{aligned}$$

□

We may use the θ-mgu instead of the σ-mgu. θ-mgu has the following property.

Lemma 95. *Given terms s and t, if $\theta = mgu_\theta(s,t)$, then for every variable X, we have $size(\theta(s)) \leq max_<(\{size(s), size(t)\})$.*

Proof. By induction on derivation $\{\langle s,t\rangle\}, [], [] \Rightarrow^* \emptyset, \theta, \sigma$. □

Now, we replace σ in the previous section by θ in our construction of $\mathbf{G}$ from the previous section. Denote the resulting sequence by $\mathbf{G}'_k(P)$, and its limit by $\mathbf{G}'(P)$.

The construction $\mathbf{G}'$ is

$$\begin{array}{rcl}
\mathbf{G}'_0(P) & = & \mathbf{F}(P) \\
\mathbf{G}'_{k+1}(P) & = & \mathbf{G}'_k(P) \cup \mathbf{F}(pair(\theta)) \text{ where there is} \\
 & & \text{a variable } X, \\
 & & \text{a function } \pi_i \circ \overleftarrow{s} \circ \pi_j \in F_X, \\
 & & \text{a function } \overrightarrow{t} \in F_{\pi_j(\mathbf{X})} \text{ such that} \\
 & & mgu_\theta(s, t) = \theta \\
\mathbf{G}'(P) & = & \bigcup_{k=0}^{\infty} \mathbf{G}'_k(P)
\end{array}$$

An immediate theorem following Lemma 95 is that the construction $\mathbf{G}'$ always terminates in finite number of iterations.

Theorem 96. *The exists an integer K such that $\mathbf{G}'_K(P) = \mathbf{G}'_{K+1}(P) = \ldots$*

Proof. The simplification only generates term functions based on terms with sizes that are smaller or equal to existing terms. Since there are only finitely many terms within a given size bound, eventually the construction will stop generating new term functions. □

4.5 Equivalent Types

4.5 Examples

In this subsection, we discuss a topic that is extremely important in the implementation of our type inference algorithm: how to simplify the types generated by our type inference algorithm to reduce the redundancy in the generated types. In order to simplify generated types, we need algorithms for finding equivalent types. If two types are equivalent, then we only need to generate one instantiation set for both of them. The goal of this subsection is not to find a complete set of simplifications that reduce the types to a minimum set, but to find out simplifications that apply to commonly seen patterns of generated types.

Let us first look at a few examples of redundant types.

Example 97.

$$Z \quad ::= \quad c \mid g(c) \mid g(Z) \tag{4.5.1}$$

In (4.5.1), $g(c)$ is redundant since the term function $g(c)$ can be generated by composing $g(Z)$ with c. To eliminate the redundancy in the this example, we need a way to find out, given a term s that appears on the right-hand side of a formal grammar rule, whether there is a another term t on the right-hand side of the formal grammar rule that is more general than s.

Example 98.

$$X \quad ::= \quad c \mid g(Z) \tag{4.5.2}$$

$$Z \quad ::= \quad c \mid g(Z) \tag{4.5.3}$$

In (4.5.2), type X is redundant since it generates exactly the same terms as type Z. To eliminate the redundancy in this example, we just need to compare two formal grammar rules and see if they are "syntactically" the same.

Example 99.

$$X \quad ::= \quad c \mid g(X) \tag{4.5.4}$$

$$Z \quad ::= \quad c \mid g(Z) \tag{4.5.5}$$

In (4.5.4), type X is redundant since it generates exactly the same terms as type Z. To eliminate the redundancy in this example, we need to compare two formal grammar rules and see if they generate the same set of terms by induction.

Example 100.

$$X \;::=\; c \mid g(Z) \tag{4.5.6}$$

$$Z \;::=\; c \mid g(X) \tag{4.5.7}$$

In (4.5.6), the type X is redundant since it generates the exactly the same terms as the type Z. In order to eliminate this redundancy, we need to compare two formal grammar rules and see if they generate the same set of terms by simultaneous induction.

Example 101.

$$X \;::=\; \ldots \mid Z \tag{4.5.8}$$

In (4.5.8), type X is redundant since it generates exactly the same terms as type Z. To eliminate the redundancy, we simply remove identify type X with type Z. We require that this kind of simplification is done for every equation. Note that we cannot simply replace all occurrences of X by Z in other formal grammar rules. For example, if we have

$$Y \;::=\; f(X, Z)$$

Replacing X by Z would make it a formal grammar rule with a (possibly) smaller set of generated terms in which all the terms must have the form $f(Z, Z)$, i.e., the two arguments of f must be the same:

$$Y \quad ::= \quad f(Z, Z)$$

4.5 Finding Redundant Terms

We use a very simple algorithm for finding redundant terms in our formal grammar. Given a term t that appears on the right-hand side of the formal grammar rule for variable X, construct a new formal grammar rule that is the same as this rule except that t is removed from the right-hand side. Replace the former rule by the latter rule in our formal grammar. We want to test if any term generate by t can still be generated for X.

We maintain a set U of unvisited pairs of variables and terms. Initially, $U = \{\langle X, t\rangle\}$. Run the procedure $redundant(U)$ defined as follows:

1. If U is empty, then return $TRUE$;
2. Pick a pair $\langle X, t\rangle$ from U.
3. For every term s that appears on the right-hand side of the formal grammar rule for variable X,
    (a) If t is an instance of s with $t = \sigma(s)$ where σ is regular, then
        i. Let $X_1, \ldots, X_n$ be all variables such that $\sigma(X_i) \neq X_i$ for $1 \leq i \leq n$, let $U' = U \cup \{\langle X_1, \sigma(X_1)\rangle, \ldots, \langle X_1, \sigma(X_1)\rangle\} \backslash \langle X, t\rangle$.
        ii. Run $redundant(U')$
        iii. If it returns $TRUE$, then return $TRUE$
        iv. If it returns $FALSE$, then continue with Step 3
    (b) If t is not an instance of s, then continue with Step 3

4. Return $FALSE$

For example, for Example 97 in the previous subsection, we want to show that $g(c)$ is redundant. We have

$$Z \ ::= \ c \mid g(c) \mid g(Z)$$

If we remove $g(c)$, we have

$$Z \ ::= \ c \mid g(Z)$$

We start with

$$U \ = \ \{\langle Z, g(c) \rangle\}$$

Pick the pair $\langle Z, g(c) \rangle$ from U, and try to see if there is a term in $c \mid g(Z)$ that is more general then $g(c)$. In the first try, we try term c, which fails. So we continue to try the next term $g(Z)$. Now $g(c)$ is an instance of $g(Z)$ with substitution $[c/Z]$. Given this substitution, we need to find out if Z is more general then c. We recursively call procedure *redundant* on the set $\{\langle Z, c \rangle\}$. In our recursive call, we have

$$U \ = \ \{\langle Z, c \rangle\}$$

Pick the pair $\langle Z, c \rangle$ from U, and try to see if there is a term in $c \mid g(Z)$ that is more general than c. In the first try, we try c, which succeeds, c is an instance of c with substitution $[]$. Since this substitution is trivial, we return $TRUE$. This return value bubbles up to the top level.

It is easy to see that this algorithm terminates since the second component of the pair always gets smaller. This algorithm can be used to eliminate redundancy like that in the Example (97) in the previous subsection, but has some limitations when it is applied to other kinds of redundancy. The main limitation is that it does not combine terms. For example, we know that

$$Z \quad ::= \quad c \mid g(c) \mid g(g(Z)) \tag{4.5.9}$$

produces exactly the same set of terms as

$$Z \quad ::= \quad c \mid g(Z) \tag{4.5.10}$$

But there is no way to use this algorithm to simplify (4.5.9) to (4.5.10).

4.5 Finding Equivalent Types

Now, we look at redundancy like that of the Example (99) and Example (100). Example (98) is a special case of this kind of redundancy.

We use the following algorithm for testing if every term generated for X is also generated for Z, given variables X and Z. Again, this is not necessarily complete in the sense that it may not return $TRUE$ if every term generated for X is also generated for Z, but it will always return $FALSE$ if there exists a term generated for X which is not generated for Z. We maintain a set U of unvisited pairs and a set V of visited pairs. Initially, $U = \{\langle X, Z\rangle\}, V = \emptyset$.

Run the procedure $instance(U, V)$ defined as follows:

1. If U is empty, then return $TRUE$

2. Pick a pair $\langle s, t\rangle$ from U

    (a) If s is a variable, then

```
        i. Let $V' = V \cup \{\langle s, t\rangle\}, U' = U \cup \{\langle s'_1, t\rangle, \ldots, \langle s'_n, t\rangle\} \backslash V'$, where $s ::=$
           $s'_1 \mid \ldots \mid s'_n$ is a formal grammar rule
       ii. Return $instance(U', V')$
   (b) Else if $t$ is a variable then
        i. For each term function $t'$ that appears on the right-hand side of the
           formal grammar rule for $t$,
            A. Let $V' = V \cup \{\langle s, t\rangle\}, U' = U \cup \{\langle s, t'\rangle\} \backslash V'$
            B. Run $instance(U', V')$
            C. If it returns $TRUE$, then return $TRUE$
            D. If it returns $FALSE$, then continue with Step 2(b)i
       ii. Return $FALSE$
   (c) Else if $t$ is an instance of $s$ with $t = \sigma(s)$ where $\sigma$ is regular, then
        i. Let $X_1, \ldots, X_n$ be all variables such that $\sigma(X_i) \neq X_i$ for $1 \leq i \leq n$,
       ii. Let $V' = V \cup \{\langle s, t\rangle\}, U' = U \cup \{\langle X_1, \sigma(X_1)\rangle, \ldots, \langle X_1, \sigma(X_1)\rangle\} \backslash V'$.
      iii. Run $instance(U', V')$
       iv. If it returns $TRUE$, then return $TRUE$
        v. If it returns $FALSE$, then return $FALSE$
   (d) Else ($t$ is not an instance of $s$)
        i. return $FALSE$
```

For example, for Example 99, we have

$$\begin{array}{rcl} X & ::= & c \mid g(X) \\ Z & ::= & c \mid g(Z) \end{array}$$

We start with

$$U = \{\langle Z, X\rangle\}$$

Pick the pair $\langle Z, X\rangle$ from U. Since Z is a variable, we enter Step 2(a)i. We construct new pairs using the term on the right-hand side of the formal grammar rule for Z and recursively run *instance* where we have

$$\begin{aligned} V &= \{\langle Z, X\rangle\} \\ U &= \{\langle c, X\rangle, \langle g(Z), X\rangle\} \end{aligned}$$

Pick the pair $\langle c, X\rangle$ from U. Since X is a variable and c is not, we enter Step 2(b)i. We want to test if any term in $c \mid g(X)$ is more general than c. In the first try, we test if c is more general than c. We recursively run *instance* with

$$\begin{aligned} V &= \{\langle Z, X\rangle, \langle c, X\rangle\} \\ U &= \{\langle c, c\rangle, \langle g(Z), X\rangle\} \end{aligned}$$

Pick the pair $\langle c, c\rangle$ from U. We enter Step 2(c)i. c is an instance of c. Therefore, we recursively run *instance* with

$$\begin{aligned} V &= \{\langle Z, X\rangle, \langle c, X\rangle, \langle c, c\rangle\} \\ U &= \{\langle g(Z), X\rangle\} \end{aligned}$$

Pick the pair $\langle g(Z), X\rangle$ from U. Since X is a variable and $g(Z)$ is not, we enter Step 2(b)i. We want to test if any term in $c \mid g(X)$ is more general than $g(Z)$. In the first try, we test if c is more general than $g(Z)$, which fails. In the second try, we test if $g(X)$ is more general than $g(Z)$. $g(Z)$ is an instance of $g(X)$ with unifier $[Z/X]$. Since $\langle Z, X\rangle$ is already in V, we run *instance* recursively where we have

$$\begin{aligned} V &= \{\langle Z, X\rangle, \langle c, X\rangle, \langle c, c\rangle, \langle g(Z), g(X)\rangle\} \\ U &= \emptyset \end{aligned}$$

It returns $TRUE$ which bubbles up to the top level.

It easy to see that this algorithm terminates, since neither U nor V contain any term that is larger in size than any term that appears in the formal grammar and eventually V will saturate.

Chapter 5

incOSHL, the Non-incremental Version

In this chapter, we look at the data structures and subroutines of the implementation of incOSHL. In the more abstract construction that we have seen so far, the $genOSHL$ function, the $simp$ function, and the m function have been given in enough details to be implemented relatively straightforwardly, the only variable left is the implementation of the d function. Basically, in our definition of the d function, we are just saying that d should choose a minimal instance of the input clauses with respect to $\leq_s$ that contradicts the current model. How to find such an instance efficiently is the main topic of the rest of the paper. The version of implementation described in this chapter is the simplest and most inefficient version. The basic idea of this version is really simple: when the algorithm tries to find a new instance, it

1. starts by looking for an instance of size 1, the smallest possible size of any instance, that contradicts the current model;

2. if it finds an instance of this size, then it returns this instance as the result of d;

3. if it cannot find an instance of this size, then it looks for an instance of size 2, that contradicts the current model;

4. so on and so forth.

The reason this version of incOSHL implementation is call "non-incremental" is that it restarts from scratch every time the global data is modified, which results in a lot of repetitive computation. Here, there are two main pieces of data that are global, one is the model, the other is the current size of which the algorithm is looking for an instance. One of the reasons why these two pieces of data are global is that they are used by almost all subroutines that we are going to define. A more essential reason is that making them global enables us to construct an incremental version of incOSHL implementation, which we will elaborate in Chapter 7.

To avoid presenting overwhelming details, we will use a Java-based pseudo-code for describing both the data structures and the subroutines of incOSHL. The pseudo-code should still be considered "algorithms" rather than actual code. However, as readers will see, they can be turned into real code fairly easily.

5.1 Some Pseudo-code Notations

In this section, we introduce some language constructs and rules for our pseudo-code that are not found in Java. We will list their syntax and give an informal explanation of their semantics.

1. As with most pseudo-code, we do not require explicit declaration of local variables. We do, however, require a subroutine's parameter types and return type to be explicitly specified, since they are considered part of the subroutine's specification.

2. We allow methods to return tuples. For example, we can define a method

```
(int , int) id2(int x, int y) {
  return (x, y);
}
```

We allow assigning a tuple returned from a method to a tuple of variables. For example, we can write

```
(x, y) = id2(a, b);
```

We use underscore "_" to represent a return value that we do not need.

```
(_, _) = id2(a, b);
```

3. We introduce the either-or control structure.

```
either <stmt> or <stmt>
```

where each "<stmt>" is replaced by a pseudo code statement. To see how an either-or statement works, let us assume that there is a stack containing program states, which we will call the "backtrack thread." When an either-or control structure is executed, a program state will be pushed onto the backtrack thread, the statement following "either" will be executed immediately after that. The program state that gets pushed onto the backtrack thread is as if the program control is immediately before the statement following "or". The "or" clause can take a guard

```
either <stmt> or if(<expr>) <stmt>
```

The effect of the guard is that the program state is saved only if "<expr>" evaluates to true at the time the either-or structure starts to execute.

4. To pop a program state from the "backtrack thread," we use the backtrack statement:

```
backtrack
```

A backtrack statement pops the top program state from the "backtrack thread" and restores that program state. As you can see, the either-or control structure and the backtrack statement are dynamically scoped, rather than lexically scoped.

5. The truncate statement

```
truncate
```

clears all program states saved. The truncate statement will not be used in our fully incremental version to be described in Section 7.2, but before that we still need it for our non-incremental version and partially incremental version.

5.2 Data Structures

In this section, we take a look at the data structures used in our incOSHL implementation. These data structure are shared by both the incremental version and the non-incremental version of incOSHL implementation. For ease of presentation, all the data structures defined here are simplified and some variable names are renamed from the actual source code to match the terminology used in this dissertation. We will also only give the interface of each class, instead of listing full implementation details such as private members and method bodies.

The first class is the SymbolId class.

```
class SymbolId {
  boolean isFuncSymbol;
  int arity;
}
```

In our algorithm, each symbol has a unique id and each id is represented by a unique SymbolId object in our pseudo-code. The “isFuncSymbol” field is true if the symbol is a function symbol (or a predicate symbol), and false if the symbol is a variable. If “isFuncSymbol” is true, then the “arity” field stores the arity of the function symbol (or the predicate symbol); if “isFuncSymbol” is false, then the “arity” field stores 0. In the real implementation, however, we use a more efficient way to handle symbolId, which basically uses a 64-bit unsigned integer to store the id. Function symbols and variables have their own value ranges. Hence the “isFuncSymbol” is implicitly encoded. The arity is also encoded by the lower bits of the 64-bit integer, given the observation that the neither the arity of a function symbol, nor the number of function symbols is likely to be large. The current version supports up to $16K$ (2^{14}) function symbols with arity up to 255 ($2^8 - 1$) and up to $4M$ (2^{22}) variable symbols.

The next class is the Symbol class.

```
class Symbol {
  SymbolId key;
  Symbol next;
}
```

A Symbol object represents the occurrence of a symbol in a term. For example, the term $f(f(a))$ is represented by three Symbol objects that represent f, f, and a, respectively, where the two occurrences of fs are represented by two different Symbol objects. The “key” field store the unique id of the symbol. Different occurrences of the same symbol have the same “key.” The “next” field of a Symbol object points to a Symbol object representing the occurrence of the symbol on its right. The “next” field of the Symbol object that represents the occurrence of rightmost symbol in a term points to “null.” For example, in term $f(a)$, the “next” field of the Symbol object that represents the occurrence of f points to the Symbol object that represents the occurrence of a and the “next” field of the Symbol object that represents the occurrence of a points to “null.” In the actual source code, the Symbol objects are not stored in linked list, but an array, and the next field is not needed any more. They also contains extra metadata about the term such as, if this symbols is a variable, whether this is the first occurrence of this variable, which are filled in as part of the preprocessing step and are used in the proof search to reduce unnecessary repeated computation.

The next class is the Term class.

```
class Term {
  Symbol firstSymbol;
  int termSize;
}
```

A Term object represents a term. Here, we use the word “term” in a more general sense which includes both first-order logic terms and atoms. The “firstSymbol” field of a Term object points to a Symbol object that represents the occurrence of the leftmost symbol of the term. For example, the “firstSymbol” field of a Term object that represents the term $f(f(a))$ points to a Symbol object that represents the first occurrence of f. The “termSize” field stores the number of symbols in the term, i.e., the value of the $size(-)$ function. In actual source code, the firstSymbol field is stored separately, as a term may have different metadata when it occurs in different places. But since we do not use any of those metadata in our pseudo-code, we just store it in the Term object, which is simpler to present. Also in actual source code, all terms are perfectly shared. A global object called the “term matrix” is used to create new terms and look up existing terms. Once references to terms are obtained, term equality is compared simply by pointer comparison.

The next class is the Literal class.

```
class Literal {
  Term atom;
  boolean isPositive;
  Literal complement;
  Clause modelClause;
}
```

A Literal object represents a literal. The “complement” field stores its complement. The “modelClause” field works as follows: in the kth iteration of our construction in Chapter 3, it points to a ground clause in T_k whose maximum literal is a complement of this literal. We only need to give a meaningful value to this field for objects representing ground literals that are false in the initial model I_0. Recall Condition 1 of the “perfect linking” property: there is exactly one clause in T_k whose maximum literal is a complement of any literal in this group. Hence it is well-defined.

The next class is the TrieNode class.

```
class TrieNode {
  SymbolId key;
  TrieNode funcChild;
  TrieNode sibling;
  TrieNode lookup(SymbolId);
  boolean available;
  Term subterm(TrieNode start);
  TrieNode insert(Symbol);
  TrieNode remove(Symbol);
}
```

A trie is a term indexing [41] data structure. Our implementation is a dictionary stored in a standard “child-sibling” tree. Each node in the tree stores a symbol and each path in the tree corresponds to a term. For example, the trie in Figure 5.2.1 stores terms $f(a)$, $g(a, a)$, and $g(b, a)$. A TrieNode object represents a node in a trie. The “key” field stores the unique id of the symbol stored in the current node. The “funcChild” field points to one of its child nodes and the “sibling” field points to its sibling node. The “lookup” method looks up a child with a unique symbol

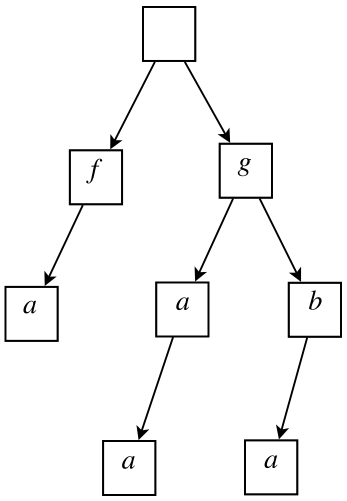

Figure 5.2.1: A Trie

id. This method will always return a TrieNode object, but the TrieNode object's "available" field may be false. The "available" field indicates whether a TrieNode object actually represent an existing trie node. The "subterm" method returns a subterm with symbols starting from "start" (exclusive) and ending at the current node (inclusive). The "insert" method inserts a new path into the trie and returns the leaf node of the path. The "remove" method removes a path from the trie and returns the leaf node of the path.

The next class is the Sub class.

```
class Sub {
  Sub push(SymbolId, Term);
  Term lookup(SymbolId);
}
```

A substitution is stored in a linked stack. A Sub object represents both a node in such a stack and the substitution represented by the portion of the stack below the node (including the node). The data structure is immutable to avoid inconsistencies during backtracking. The "push" method takes in a unique symbol id (which must be that of a variable) and a term and returns a new Sub object representing a new substitution which is the same as the substitution that the current Sub object represents except it maps the unique symbol id to the term. For example, if we have a Sub object that represents the substitution $[f(a)/X][a/Y]$, then by invoking the push method on symbol Y and term $g(a)$, we obtain a Sub object that represents the substitution $[f(a)/X][g(a)/Y]$.

The next class is the PerfectlyLinkedSet class.

```
class PerfectlyLinkedSet {
  Clause findClauseToDelete(Set);
  void deleteClause(Clause);
  void addClause(Clause);
}
```

The “addClause” method adds a clause to the set, which may make the set no longer perfectly linked. The “findClauseToDelete” method finds a clause that needs to be deleted towards restoring the perfect linking property of the set. The “deleteClause” method actually deletes a clause from the set.

The next class is the Clause class.

```
class Clause {
  Literal[] literals;
  void sort();
}
```

A clause object contains an array of literals. The “sort” method does an in-place sort of the array of literals in descending order, so that the first literal is always the maximum literal.

Finally, we look at class ClauseTreeNode.

```
class ClauseTreeNode {
  boolean isLeaf;
  Literal literal;
```

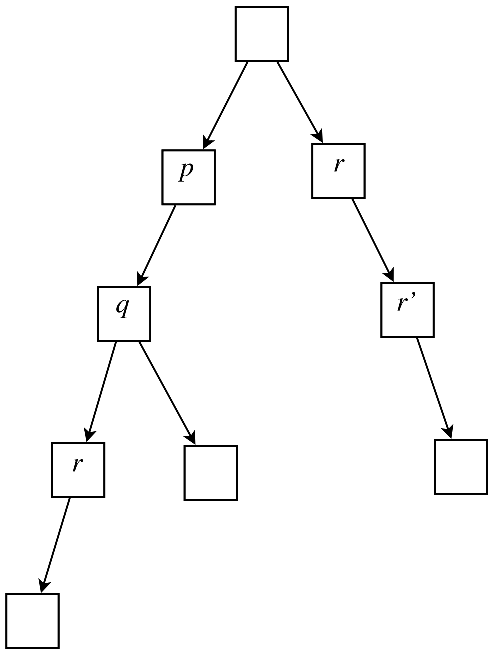

Figure 5.2.2: A Clause Tree

```
  ClauseTreeNode[] subtrees;
}
```

Clause trees are used to store input clauses. Each ClauseTreeNode object represents a node in a clause tree. The “isLeaf” field indicates whether this is a leaf node. The “literal” field stores a literal, except in root and leaf nodes, where “isLeaf” is null. A clause is represented by a path from the root of the clause tree (exclusive) to a leaf (exclusive) in the clause tree. [1] The “subtrees” array stores pointers to root nodes of all subtrees. An example of the clause tree containing clauses $\{p, q\}$, $\{p, q, r\}$, $\{r, r'\}$ is shown in Figure 5.2.2. Again, in the actual source code, ClauseTreeNode objects contain extra metadata about the clauses, which are filled in as part of the preprocessing step and are used in the proof search to reduce unnecessary repeated computation.

[1] In our implementation, we do not actually create objects for the leaf nodes. The “isLeaf” node indicates whether there is a leaf under the current node.

5.3 Subroutines

In this section, we look at the subroutines of the non-incremental version of incOSHL. There are four main subroutines:

1. "lookup" finds an instance of a term from a trie. This subroutine implements the unification algorithm. It takes in a term and a trie of ground terms. It returns, if possible, a substitution such that the returned substitution instantiates the input term to a term in the input trie.

2. "inverseLookup" finds an instance of a term not in a trie. This subroutine implements the disunification algorithm [32]. It takes in a term and a trie of ground terms, and returns, if possible, a substitution, such that the returned substitution instantiates the input term to a term *not* in the input trie.

3. "traverse" traverses a ClauseTree object, and calls "lookup" and "inverseLookup" to generate an instance of a clause stored in the ClauseTree object which contradicts the current model.

4. "updateModel" updates the current model and generates a new model that makes a contradicting instance true.

We assume that

1. There is a global variable "globalSizeLimit" which is the current size of which the algorithm is trying to find a contradicting instance.

2. There is a global variable "trieRoot" which is the root node of the trie that indexes the current model;

3. There is a global variable "perfLinkedSet" that represents T_k, the current set of ground instances.

5.3 Finding an Instance of a Term from a Trie

The "lookup" subroutine looks like:

```
(TrieNode, Sub, int)
lookup(int n, TrieNode node, Symbol symbol, Sub sub) {
  if( symbol == null ) {
    return (node, sub, n);
  } else if( symbol.key.isFuncSymbol ) {
    nodeNew = node.lookup(symbol.key);
    if( ! nodeNew.available ) {
      backtrack;
    } else {
      return lookup(n, nodeNew, symbol.next, sub);
    }
  } else { // symbol is a variable
    subterm = sub.lookup(symbol);
    if( subterm == null ) {
      (nodeNew, retterm, nNew) = branch(n, cnode, cnode, 1);
      subNew = sub.push(symbol.key, retterm);
      return lookup(nNew, nodeNew, symbol.next, subNew);
    } else {
      nInput = n + subterm.termSize - 1;
      if( nInput > globalSizeLimit) {
        backtrack;
      }
      (nodeNew, subNew, nNew) =
        lookup(nInput, node, subterm.firstSymbol, sub);
      return lookup(nNew, nodeNew, symbol.next, subNew);
    }
  }
}
```

The “lookup” subroutine looks up an instance of a term from a trie. It takes in four parameters:

1. “n” is a lower bound to the size of any term in the trie that matches the input term. If “n” is greater than “globalSizeLimit” then it means that even if we continue further down, no more instances with size “globalSizeLimit” can be found. The initial value of “n” should be the size of the input term.

2. “node” is the current trie node. It indicates our progress down the trie. The nodes between the root (exclusive) and the current nodes (inclusive) have already been matched with a prefix of the input term.

3. “sub” is the substitution generated from matching that prefix with the trie.

4. “symbol” is the next symbol in the input term to be matched with a children of the current trie node.

The “lookup” subroutine is recursive in two ways. First, after matching a symbol with a node, it recursively invokes itself to match the next symbol. Second, if a variable has already been bound to a term, then when the same variable appears again, the subroutine recursively invokes itself using the term that is bound to the variable as the new input term, and the current node as the new starting node; when the recursive invocation returns, it resumes from the node where the recursive invocation finishes. For example, if the input term is $g(X, X)$, then we know when the second X is encountered, X must have been bound to some term. The subroutine recursively invokes itself, looking for an instance of the term that X is bound to from the current node. Since in our algorithm, the trie contains ground terms only, this second kind of recursion has at most two levels. But this algorithm can be easily generated to tries containing non-ground terms using “versions.” (See source code for more details about versions)

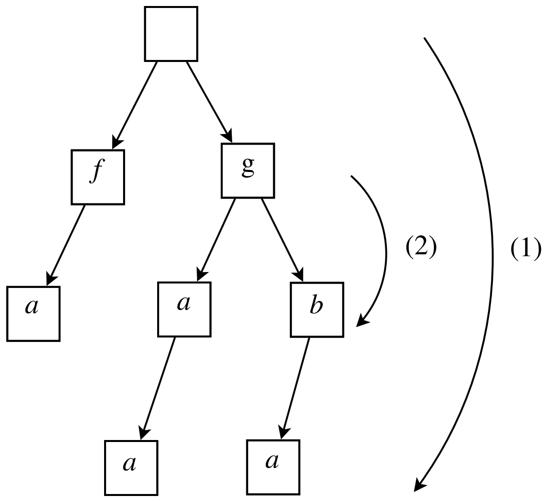

Figure 5.3.1: TrieNode Objects Returned by the “lookup” Subroutine

The “lookup” subroutine returns three return values:

1. The first one is the node in the trie where the an instance of the input term is found. The node is not necessarily a leaf node of the trie if “lookup” is not called on “trieRoot.” This is illustrated in Figure 5.3.1. (1) If we start from the root, and the input term is $g(X, a)$, then an instance of the input term is $g(b, a)$. The subroutine ends at the rightmost node containing a. (2) If we start from the node containing g, and the input term is X, then an instance of the input term is b, a subterm of $g(b, a)$. The subroutine ends at the node containing b.

2. The second one is the new substitution.

3. The third one is the new lower bound. If this call is a recursive call, then this return value reflects the lower bound of the top level call.

The “branch” subroutine, invoked in the “lookup” subroutine, finds a subterm starting from the current node.

```
(TrieNode, Term, int)
branch(int n, TrieNode start, TrieNode curr, int np) {
  if( numberOfPlaceholders == 0 ) {
    return (curr, curr.subterm(start), n);
  } else {
    return branchTryTargetNodes(n, start, curr, 1);
  }
}


(TrieNode, Term, int)
branchTryTargetNodes(int n, TrieNode start, TrieNode curr, int np) {
  either {
    nInput = n + curr.symbol.arity;
    if( nInput > globalSizeLimit ) {
      backtrack;
    } else {
      return branch(nInput, start, curr, np + arity - 1));
    }
  } or {
    if(curr.sibling == null) {
      backtrack;
    }
    return branchTryTargetNodes(n, start, curr.sibling, np);
  }
}
```

The "branch" subroutine takes in four parameters. "n" is a lower bound to the size any term in the trie that matches the input term. "start" is the starting node. "curr" is the current node. "np" is the number of placeholders.

"branch" simply generates all subterms starting from the start term. The nodes between the starting node (exclusive) and the current nodes (inclusive) is a prefix of terms that it generates. For example, as shown in Figure 5.3.2, suppose that it starts from the root, and one placeholder. It can go down to the node containing

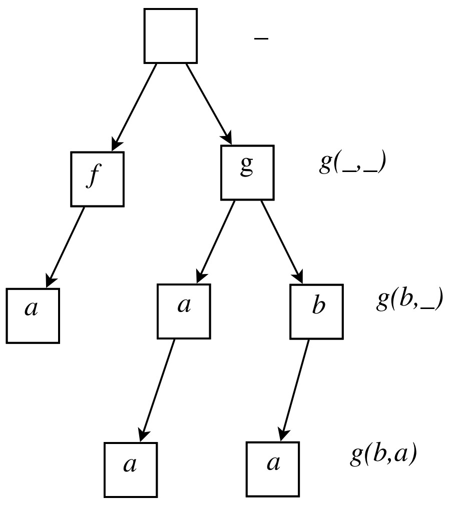

Figure 5.3.2: A Example Of the "branch" Subroutine

g, generate the prefix g, and place it in the placeholder. Since g has arity 2, it generates two new placeholders. Next, it can go down to the node containing b, generate the prefix b, and place it in one of the new placeholders. Since b has arity 0, it does not generate any new placeholders. Next, it can go down to the node containing a, generate the prefix a, and place it in the other new placeholder. Since a has arity 0, it does not generate any new placeholders. It stops there and returns the generated term.

The nondeterminism for choosing which child node to go down is encapsulated in the either-or control structure in the "branchTryTargetNode" subroutine.

5.3 Finding an Instance of a Term That Is Not in a Trie

We view the problem of finding an instance of a term that is not in a trie as a problem of finding an instance of a term in the complement trie. The complement of a trie is a trie containing all terms not in the former trie. For example, the complement trie of the trie in Figure 5.2.1 looks like that in Figure 5.3.3. We cannot show the whole complement trie since it has an infinite number of nodes. The complement of a finite set in a infinite set is infinite. The solution is that we

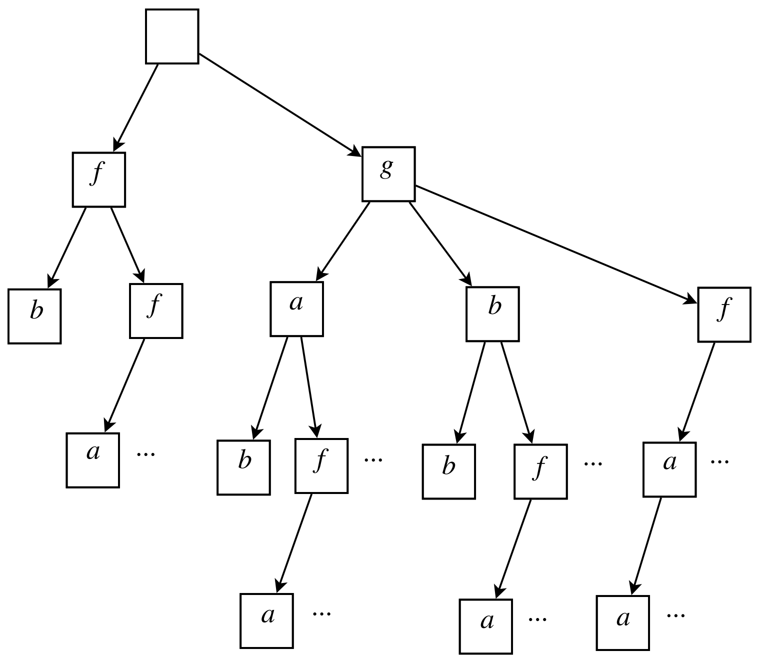

Figure 5.3.3: A Complement Trie

grow the complement trie in a lazy manner. The “lookupInverse” subroutine looks like:

```
(TrieNode, Sub, int)
lookupInverse(int n, TrieNode node, Symbol symbol, Sub sub) {
  if( symbol == null ) {
    return (node, sub, n);
  } else if( symbol.key.isFuncSymbol ) {
    nodeNew = node.lookUp(symbol.key);
    return lookupInverse(n, nodeNew, symbol.next, sub);
  } else { // symbol is a variable
    subterm = sub.lookup(symbol);
    if( subterm == null ) {
      (nodeNew, retterm, nNew) = branchInverse(n, node, node, 1);
      subNew = sub.push(symbol.key, retterm);
      return lookupInverse(nNew, nodeNew, symbol.next, subNew);
    } else {
      nInput = n + subterm.termSize - 1;
      if( nInput > globalSizeLimit) {
        backtrack;
      }
      (nodeNew, subNew, nNew) =
        lookupInverse(nInput, node, subterm.firstSymbol, sub);
      return lookupInverse(nNew, nodeNew, symbol.next, subNew);
    }
  }
}
```

The key differences between this subroutine and the "lookup" subroutine are highlighted in the code. First of all, when the end of the term is reached, it returns the substitution and the corresponding TrieNode object anyway. If the TrieNode object is available, then the caller of this function should discard the result. Second, instead of calling the "branch" subroutine, it calls the "inverseBranch" subroutine.

The "branchInverse" subroutine considers all possible terms that a variable can be instantiated to. It depends on a array "functionSymbols" of all possible function symbols generated when parsing the input clauses. The "inverseBranch" subroutine looks like:

```
(TrieNode, Term, int)
branchInverse(int n, TermCacheNode currTCN, TrieNode node,
  int nPlaceholders) {
  if( nPlaceholders == 0 ) {
    return (node, currTCN.term, n);
  } else {
    return branchInverseTryFuncSymbols(n, currTCN, node,
      nPlaceholders, 0);
  }
}
```

```
(TrieNode, Term, int)
branchInverseTryFuncSymbols(int n, TermCacheNode superTCN,
  TrieNode node, int nPlaceholders, int i) {
  either {
    s = functionSymbols[i];
    newTCN = superTCN.lookup(s);
    nodeNew = node.lookup(s);
    nInput = n + s.arity;
    if(nInput > globalSizeLimit) {
      backtrack;
    } else {
      return branchInverse(nInput, newTCN, nodeNew,
        nPlaceholders + s.arity - 1);
    }
  } or if(i + 1 != numberOfFunctionSymbols ) {
    return branchInverseTryFuncSymbols(n, superTCN, node,
      nPlaceholders, i + 1);
  }
}
```

5.3 Traversing a Clause Tree

Now we take a look at how to traverse a clause tree to find an instance of a clause that contradicts the current model. Recall that we assume that there is a global variable “trieRoot” which is the root node of the trie that indexes the current model.

```
(ClauseTreeNode, Sub)
traverse(ClauseTreeNode node, Sub subInit) {
  if(node.literal == null) {
    retsub = subInit;
  } else if(node.literal.isPositive) {
    (retnode, retsub, _) = lookupInverse(node.literal.atom.termSize,
      trieRoot, node.literal.atom.firstSymbol, subInit);
    if( retnode.available ) {
      backtrack;
    }
  } else {
    (_, retsub, _) = lookup(node.literal.atom.termSize,
      trieRoot, node.literal.atom.firstSymbol, subInit);
  }
  if(node.isLeaf) {
    return (node, retsub);
  } else {
    return traverseSubtrees(node, retsub, 0);
  }
}


(ClauseTreeNode, Sub)
traverseSubtrees(ClauseTreeNode node, Sub subNew, int index) {
  either
    return traverse(node.subtree(index),subNew);
  or if(index + 1 < node->degree)
    return traverseSubtrees(node, subNew, index + 1);
}
```

The “traverse” subroutine tries to “lookup” or “lookupInverse” the literal stored in current node. If the literal is null, it means the current node is the root node of the clause tree and we should directly traverse its subtrees. The “traverseSubtrees” subroutine tries to traverse all subtrees of the current node. Again, the non-determinism is encoded in the either-or structure of the “traverseSubtrees” subroutine.

5.3 Updating the Model

Recall that we assume that there is a global variable “perfLinkedSet” that represents T_k, the current set of ground instances. The “updateModel” method tries to update the “perfLinkedSet” so that it can generate a model and returns whether this is possible.

```
boolean
updateModel(ClauseTreeNode node, Sub sub) {
  gc = node.instance(sub);
  while(gc.literals[0].modelClause != null) {
    gc = gc.orderResolve(gc.literals[0].modelClause);
    if(gc.literals.length == 0) {
      return false;
    }
  }
  perfLinkedSet.addClause(gc);
  trieRoot.insert(a.literals[0].atom.firstSymbol);
  while((c = perfLinkedSet.findClauseToDelete()) != null) {
    trieRoot.remove(a.literals[0].atom.firstSymbol);
    perfLinkedSet.deleteClause(c);
  }
  return true;
}
```

This subroutine is basically a direct implementation of the *simp* function of in-cOSHL. First, it does ordered-resolution. Next, it adds the resolvent to the current set of ground clauses. Then, it deletes some clauses to make the new set of ground clauses perfectly linked. Note that the “findClauseToDelete” method, the detail of which we did not show, should never delete “gc,” the newly added clause. This has been taken care of in the actually source code.

Now, recall that we assume that there is a global variable “clauseTreeRoot” that stores the root of our clause tree. The main loop looks like:

```
boolean
mainLoop() {
  for(;;) {
    globalSizeLimit = 1;
    for(;;) {
      truncate;
      either {
        (noderet, subret) = traverse(clauseTreeRoot, new Sub());
        break;
      } or {
        if(saturated()) {
          return true;
        }
        globalSizeLimit++;
      }
    }
    if( !updateModel(noderet, subret) ) {
      return false;
    }
  }
}
```

The “mainLoop” iteratively generates instances that contradict the current model and updates the model. It returns whether the input clauses are satisfiable. If it returns false, then it means that the subroutine has found an unsatisfiable set of ground instances of the input clauses. If it returns true, the it means that the subroutine has found a model for the input clauses.[2] When generating instances, it starts with "globalSizeLimit", the variable that stores the upper limit of the size of a clause that can be generated, equaling 1. If no instance can be generated under the size limit, then it first tests if the instance generation process has saturated, i.e., no instance can be generated any more (satisfiable). If the answer is yes, then it returns true; otherwise, it increments the variable and retries. When an instance is generated but the model cannot be updated (unsatisfiable) any more, it returns false.

[2]Since validity in classical first-order logic is semidecidable, this does not happens for every satisfiable set of input clauses.

Chapter 6

A Language-based Approach to Efficiency

We have discussed details of our non-incremental version of incOSHL. The remaining problem is how to implement them efficiently. We have mentioned in the previous chapter about an incremental version of our incOSHL algorithm. Before showing the details of it, we would like to introduce the STACK EL, an embedded language written mainly using C/C++ macros, that enables a maintainable implementation of the incremental version of incOSHL. The main rationale of providing an embedded language instead of directly implementing the incremental version in C is to improve the efficiency of development and maintenance of our implementation. In our experience with implementing the Java version of OSHL-S, the complexity of a direct implementation is too high for a one person project, not to mention ease of maintenance. Since theorem provers, like any other software, are constantly evolving, ease of development and maintenance is a key factor in the successful evolution of a theorem prover. We followed the guideline of stratification in making this design decision, encapsulating all the common, low-level operations into a generic embedded language, and leaving only the high-level, prover-specific operations to the prover-specific code. As a result, both the embedded language and the theorem prover are made easier to test, debug, and modify.

The STACK EL is an embedded language which implements the language constructs used in the algorithms in Chapter 5. In particular, it provides in the embedded language itself constructs such as function declaration, function definition, function call, function return which parallel those of the host language but support the additional features of the STACK EL such as saving and restoring program states and global mutable data dependency.

In this implementation of the STACK EL, we make use of one of the oldest features in the C Programming Language – macros. There are many debates around whether one should or should not use macros, especially around the issue of hygienic macros. Using macros allows the STACK EL to provide an embedded language that looks and feels like a standalone programming language, while offering the following benefits:

- It is fully interoperable with C/C++.
- It is statically typed.
- STACK EL code is C/C++ code and is as portable as any C/C++ code. It does not require any external tools or non-standard runtime library. This is extremely important if one wants to run the code on a server without full administrator privileges.

6.1 The Concrete Syntax of the STACK EL

In this section, we show the concrete syntax of the STACK EL as embedded in C/C++. We follow the C/C++ convention that all-capitalized identifiers are macros.

6.1 Function Declaration

The syntax for function declaration is as follows

```
#define LEVEL_<function name> <level>
PROTO(<function name>,
  (<parameter 1 type>, ..., <parameter m type>),
  <return value 1 type>, ..., <return value n type>)
```

It declares a function with name "<function name>" at level "<level>" with m parameters and n return values, where m and n can be up to 10 in the current implementation. The level of a function is used to allow functions on different levels to see each other's stack frame. This is useful, for example, to avoid passing lots of constant values around when operations in a nested function call need to access variables declared in the function that make the nested function call. The main purpose of the "PROTO" macro is to generate a list of definitions that can be used for typechecking.

As an example of function declaration,

```
#define LEVEL_lookup 1
PROTO(lookup,
  (int, Trie *, PMVMInstruction *, Sub *),
  Trie *, Sub *, int)
```

declares a lookup function at level 1, with four parameters and three return values.

6.1 Function Definition

The syntax for function definition looks like:

```
#define FUN <function name>
  PARAMS(
    <parameter 1 type>, <parameter 1 name>,
    ...,
    <parameter m type>, <parameter m name>)
  RETURNS(<return value 1 type>, ..., <return value n type>)
  BEGIN
  DEFS(
    <local variable 1 type>, <local variable 1 name>,
    ...,
    <local variable l type>, <local variable l name>)
    <statements>
  END
```

It defines a function with name “<function name>” with m parameters, n return values, and l local variables, where m, n, and l can be up to 10 in the current implementation. For example,

```
#define FUN lookup
  PARAMS(
    int, n,
    Trie *, curr,
    PMVMInstruction *, inp,
    Sub *, sub)
  RETURNS(Trie *, Sub *, int)

  BEGIN
  DEFS(Term *, subterm,
    PMVMInstruction *, varInp,
    Trie *, varNode)
    ...
  END
```

defines a function named “lookup.”

6.1 Accessing Variables

To access a parameter or a local variable from the current stack frame, either for l-value or r-value, use the following syntax:

```
VAR(<local variable name or parameter name>)
```

For example, in our “lookup” function, we can use

```
VAR(varNode)
```

and

```
VAR(curr)
```

as normal C/C++ expressions.

6.1 Function Call

To call a function, use the following syntax:

```
FUNC_CALL(<function name>,
  (<argument 1>, ..., <argument m>),
  <l-value 1>, ..., <l-value n>)
```

where the l-values are used to store the return values. For example,

```
FUNC_CALL(lookup,
  (1, trieRoot, inp, new Sub()),
  nodeNew, subNew, nNew)
```

In this example, we have also shown that we can mix and match C/C++ elements with the STACK EL, here “1” is a C/C++ integer literal, “trieRoot”, “inp”, “nodeNew”, “subNew”, and “nNew” are all C/C++ variables, and “new Sub()” is a C++ expression.

To tail-call a function, use the following syntax:

```
FUNC_TAIL_CALL(<function name>,
  (<argument 1>, ..., <argument m>))
```

No l-value is provided to store the return values. The macro directly passes the return value on to the caller of the current function. The number of return values of the function being called and their types should match those of the calling function. For example, if we are within the "lookup" function, then we can write:

```
FUNC_TAIL_CALL(lookup,
  (1, trieRoot, inp, new Sub()))
```

To return from a function, use the following syntax:

```
RETURN(<return value 1>, ..., <return value m>)
```

Again, the number of return values and their types should match those of the function from which it returns. For example, if we are within the "lookup" function, then we can write:

```
RETURN(trieRoot, inp, new Sub())
```

6.1 Working with Program States

To save a program state, use the following syntax:

```
SAVE_STATE(<return label>)
```

The "<return label>" is a program label from where the program execution should be resumed when the program state is restored. For example,

```
  SAVE_STATE(rp)
rp:
  ...
```

The return label does not have to follow immediately after the SAVE_STATE macro.

To restore a program state, use the following syntax:

```
RESTORE_STATE
```

To truncate, use the following syntax:

```
TRUNCATE_STACK
```

Our higher-level pseudo-code constructs can be translated into our lower level macros as follows: The either-or control structure and the backtrack statement are implemented using SAVE_STATE and RESTORE_STATE as follows:

```
either <statement 1> or <statement 2>
```

is implemented by

```
  SAVE_STATE(rp)
  <statement 1>
  goto cont;
rp:
  <statement 2>
cont:
```

```
either <statement 1> or if(<expr>) <statement 2>
```

is implemented by

```
  if(<expr>)
    SAVE_STATE(rp)
  <statement 1>
  goto cont:
rp:
  <statement 2>
cont:
```

and

```
backtrack
```

is implemented by

```
RESTORE_STATE
```

6.1 Additional Features

There is a more powerful version of SAVE_STATE with declares dependency of program states on global mutable data and anchored program states:

```
SAVE_STATE_WITH_THREAD_DEPENDENCY_AND_ANCHOR(<return label>,
  <thread object>, <thread id>,
  <dependency object>, <dependency type>,
  <anchor>)
```

1. Instead of having one backtrack threads, we have multiple backtrack threads. The current thread is where we find program states to restore but we can switch to other threads if necessary. The additional parameters "<thread object>", and "<thread id>" are used for this purpose.

2. Instead of making every program state saved in every thread available for RESTORE_STATE, we may say that some program states are unavailable based on the state of certain global objects. The additional parameters "<dependency object>", and "<dependency type>" are used for this purpose.

3. "<anchor>" indicates whether this state is anchored. An anchored program state is a program state which does not get popped from the current backtrack stack, even after it is restored.

To declare a data dependency for all future actions, use the following syntax:

```
SAVE_DATA_DEPENDENCY
```

We will elaborate these addition features in Section 6.2.4 and Section 6.2.5.

6.2 The implementation of the STACK EL

The implementation of the STACK EL is based on the idea of the Spaghetti stack (or heap-allocated stack frames [4], or cactus stack [16], or saguaro stack, or in-tree). Spaghetti stacks have been used in various programing language runtime implementations, such as Scheme, Standard ML of New Jersey, and Interlisp [10].

In the STACK EL, each function is also a coroutine, which can yield/resume program control anywhere in the function. It has been shown [19] that the optimal strategy of implementing such coroutines is to allocate stack frames of "real coroutines" from the heap, while allocating stack frames of "fake coroutines" – coroutines that never yield/resume control from a stack. The implementation of the STACK EL takes advantage of the tight integration with C/C++, a benefit of embedding, and make the following distinction: "real coroutines" are written using the EL, which are allocated on a memory region, a heap-like data structure, "fake coroutines" are written in C/C++, which, naturally, are allocated on the system stack. In additional to the basic spaghetti stack, STACK EL has an important feature which is global mutable data dependency. This feature enables the efficient reuse of saved program state, even if the global mutable data has been changed.

6.2 Memory Management Using Memory Regions

A memory region is a low-level data structure that is used extensively in the current implementation of incOSHL. It has a few key capabilities that make it a better choice then the system provided malloc/free. A memory region consists of a linked list of nodes. The metadata of the memory region, such as the total size and the pointer to the first node, are stored on the first node. Similarly, the metadata about each node is stored in the node itself. There is no extra allocation except for nodes. This way, we can make the size of a node to be a multiple of the size of a operating system memory page and get relatively high performance when allocating and deallocating nodes. If all objects allocated in a memory region are of the same, fixed-size type, then the memory region can also be used as an object pool.

A memory region may grow by allocating new nodes if it run out of space. Objects allocated in a memory region can be deleted, but the memory that they occupy is not reclaimed until one of the following happens:

1. The whole memory region is deallocated.
2. In memory regions that are used as object pools, a new object is created at the location of the previous freed object.

The structs for a node and a region are:

```
struct region_node {
  unsigned char *block; // pointer to memory block
  size_t size; // size of the memory block in bytes
  size_t used; // used bytes of the memory block
  struct region_node *next; // pointer to the next region
};

struct region {
  size_t alloc_size; // size allocated
  struct region_node *head, // pointer to the first node
  struct region_node *active; // pointer to the last node
  struct region_desc *free; // pointer to a free list
};
```

Most of the members of these structs are quite straightforward. The “free” member of the “region” struct is used when the region is used as an object pool where “free” points to the free list.

Our implementation uses a memory region in one of three following ways:

1. Use it as a one time allocation buffer. The objects allocated in the memory region remain allocated for the whole cycle of the theorem prover. Objects allocated this way are those that are used globally and generated only during the preprocessing phase.

2. Use it as a garbage collected buffer. The objects allocated in the memory region cannot be “freed” and its space will be recycled when the garbage collector is called explicitly and programmatically. Most objects, including stack frames of the coroutines, are allocated this way.

3. Use it as an object pool. The objects allocated in the memory region are of the same type. The objects can be “freed” and their space will be recycled when a new object is created at the location of the previously freed object.

6.2 The STACK EL Stack

A STACK EL stack, similar to a Spaghetti stack, can be viewed as a tree. Each path of the tree corresponds to a separate stack, with the leaf node being the top of the stack. In a sequential setting, there is alway one path that is currently active. Other paths represent stacks used by coroutines which have yielded control. The leaf of the active path is the active stack frame. The tree structure of a STACK EL stack allows different stacks to share a common lower portion of the stack, avoiding the need to copy the whole stack when saving and restoring program states.

In our STACK EL, each either-or control structure causes the STACK EL stack

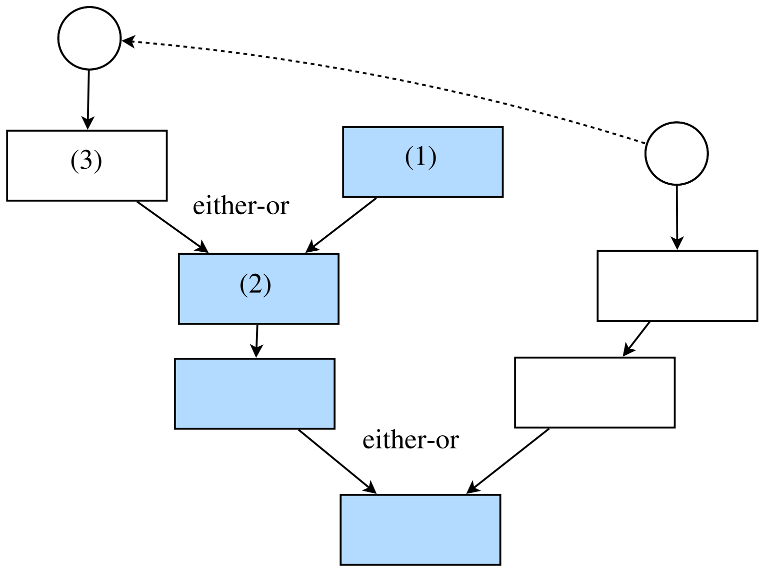

Figure 6.2.1: A STACK EL Stack

to save the current program state and create a new branch from the active stack frame. Multiple encounters of one or more either-or control structure create a tree like structure where the tree nodes are stack frames. For example, as shown in Figure 6.2.1, where the rectangles are stack frames, the arrows with solid lines pointing from one stack frame to another stack frame indicate that the former is created while the latter is being active. The highlighted path represents the active stack. Other inactive stacks share certain nodes with the active stack.

In terms of memory management, all stack frames are allocated in a memory region designated for stack frames only. This memory region is garbage-collected programmatically when the memory is near full. The garbage collector uses a very simple generational copy-based algorithm.

In the absence of either-or control structures, the STACK EL stack acts as if it is a normal stack, i.e., it grows with function call and retracts with function returns. When it grows, it allocates from the stack frame memory region. When it retracts, it does nothing. The garbage collection subroutine starts from the active frame, follows the arrows with solid lines, keeps all reachable frames, and deletes all unreachable frames. This way, simply leaving a frame unreferenced is sufficient for later reclaiming its memory space. In the presence of either-or control structures, the STACK EL stack cannot always retract with function returns. For example, in Figure 6.2.1, when a function return occurs in stack frame (1), the stack frame should be garbage-collected, but stack frame (2) should not always be garbage-collected, since it is shared by the path to the left of the current active path. Stack frame (3) should not be garbage-collected, either, since it is part of a saved program state. Without extra pointers, stack frames like (2) and (3) can easily become unreferenced. This is where the objects represented by circles come into play. The SAVE_STATE action marks the current stack frame as still "in use" by creating a special type of object, denoted by a circle in the figure, representing a saved program state. The special object contains a pointer to the currently active stack frame, so that no matter where the active stack frame changes to, this stack frame will always be referenced by the special object, and will not be eligible for garbage collection, until a RESTORE_STATE action deletes the special object. (The dotted line signifies that the two special objects are in the same "thread.")

To implement the special object, we have the following data structure (the struct is simplified and names are renamed to match the terminology of this dissertation)

```
struct ProgramState {
  void *programPointer;
  StackFrame *stackFrame;
  ProgramState* prevInThread;
  bool anchor;
  ...
}
```

where "programPointer" points to a label in the program from where the program execution should resume, "stackFrames" points to the active stack frame when the state is saved (and restored), “prevInThread” points to a ProgramState object representing the previous program state in the thread, and “anchor” indicates whether this program state is an anchored program state. All ProgramState objects are allocated in a designated memory region which is used as an object pool.

6.2 Tail-Call Optimization on the STACK EL Stack

The implementation of function calls should take tail-call optimization into consideration, since there are lots of tail-calls in our algorithm.

A tail-call is a special function call that has the form:

```
return <function call>;
```

i.e., the return value of the next level of function call is returned immediate from the current level of function call – the function call is the last operation before the current function returns. As we have seen, there are lots of tail-calls in our subroutines. Therefore, it is extremely important for our implementation of the STACK EL stack to support tail-call optimizations.

In a linear stack, tail-call optimization includes two parts:

1. reusing the caller’s stack frame for that of the callee and
2. returning directly from the callee to the caller’s caller.

This is illustrated in Figure 6.2.2. On the top is a regular function call. When the function call is initiated, a new stack frame is allocated for the callee. When the callee returns, the callee’s stack frame is deleted, and the active stack frame switches back the caller’s stack frame. On the bottom is a tail-call. When the function call is initiated, a new stack frame is created at the location of the current stack frame, overwriting the stack frame of the caller. When the callee returns, the active stack frame switches directly to that of the caller’s caller, avoiding the

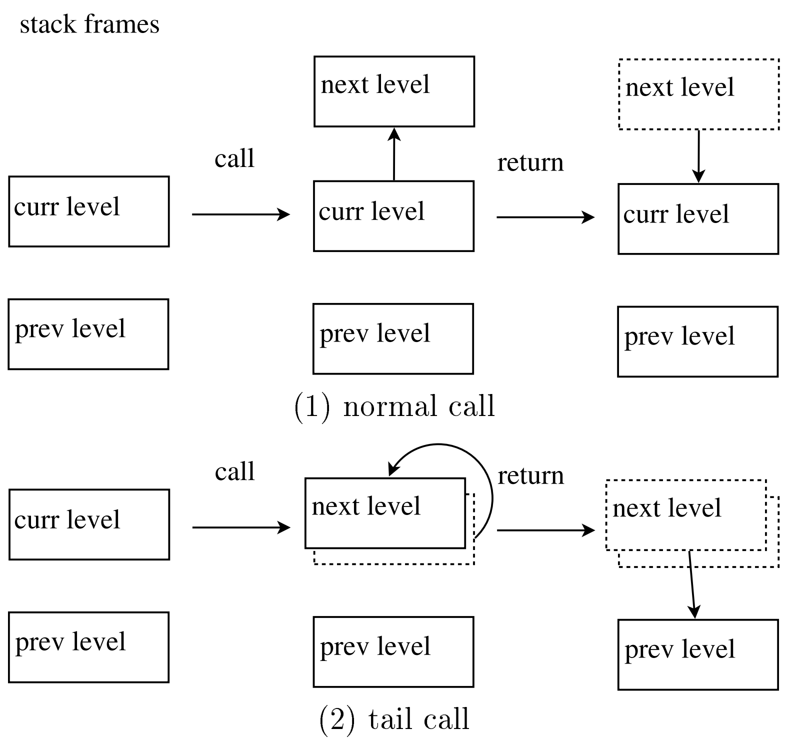

Figure 6.2.2: Normal vs Tail Call

caller altogether. In fact, there is a well-known technique in the functional programming community called continuation-passing style (CPS) transformation which transforms a functional program with both tail and non-tail calls into one that is semantically equivalent but only has tail-calls. This technique has been applied in compilers for functional programming languages to eliminate the need for a runtime stack altogether.

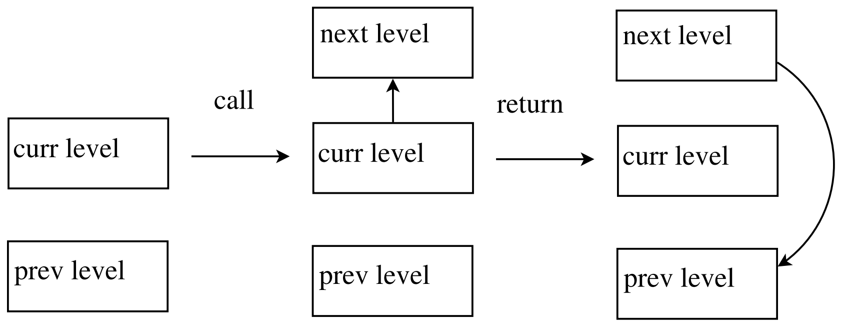

Figure 6.2.3: Tail Call Optimization in STACK EL Stacks

Now if we reexamine tail-call optimization in the context of our STACK EL stack, we find that we can do optimization 2 but not optimization 1. To see why, recall that in our STACK EL stack, which is a tree, different paths can share the lower portion of stack frames. Overwriting any stack frames in that portion from one path may affect another path that shares that stack frame. Therefore, in our implementation, we only do optimization 2, as shown in Figure 6.2.3.

6.2 Threads

One of the motivations of saving and restoring program states is to reduce repeated computation. However, a Spaghetti stack based implementation only guarantees the correctness of data stored on the stack. Any global mutable data stored outside the stack, such as the trie in our algorithm, are not covered. Sometimes, the global mutable data are important to achieve reasonable performance and passing those data around as function arguments or making them immutable is simply not an option.

Threads provide a mechanism to deal with changes in the global mutable data and help manage the dependency of program states on global mutable data. The general idea of threads is that we provide a form of implicit data dependency from program state to the thread it is in. It is called implicit because the availability of a program state is not directly tested based on the special object storing the program state, but is implicitly determined by switching between threads. The program states in the current thread are available, the program states in other threads are not available.

The STACK EL supports multiple threads. Multiple threads are useful when we want to save some program state that is reserved for the future. One example of program states like this is those when the "lookup" subroutine finds out that

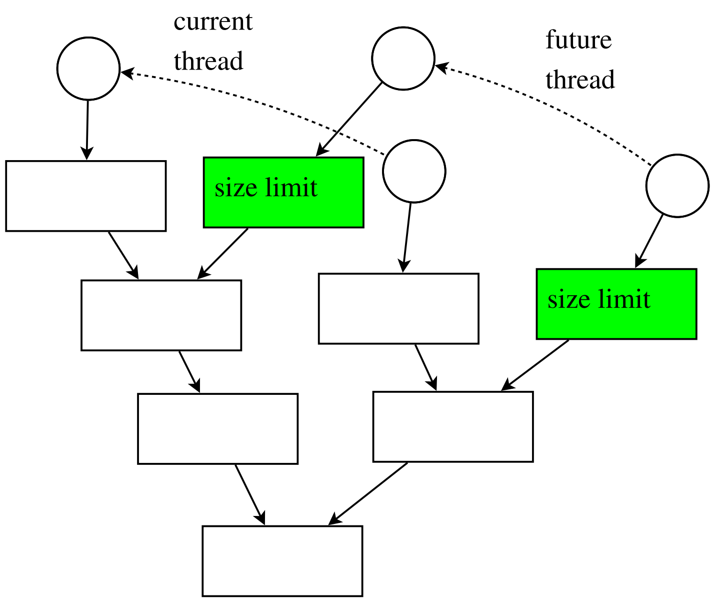

Figure 6.2.4: Threads

the lower bound of any instance that can be generated is larger than the current “globalSizeLimit.” In our non-incremental version, such program states are not saved and the algorithm simply backtracks. A more efficient way to deal with these program states is to perform a SAVE_STATE, storing them in threads reserved for the future when “globalSizeLimit” has been raised to allow instances with larger sizes. The idea of future threads is illustrated in Figure 6.2.4.

In our pseudo code, we introduce a variant of the the either-or control structure which is:

```
either <stmt> or in_thread[<id>] <stmt>
```

where “<id>” is an id of a thread. Any program state stored in a thread other than the active thread is reserved for future use. Accompanying this statement is:

```
switch_to_thread(<id>)
```

which allows us to switch to a certain thread with id “<id>”. Another feature that will be used is an anchored program state:

```
anchored
```

Normal program states are popped and discarded when they have been restored once. Anchored program states are program states that will not be popped or discarded even if they have been restored.

To implement multiple threads, we extend the ProgramState struct from Section 5.2 as follows:

```
struct ProgramState {
  void *programPointer;
  StackFrame *stackFrame;
  ProgramState* prevInThread;
  bool anchor;
  int threadId;
  ...
}
```

The added member, as highlighted, stores the id of the thread this program state is in. There is an array of “thread pointers” pointing to the top program state in each thread. There is also an “active thread” pointer, which is global across multiple threads, that points to the top program state in the active thread. Saving a program state to or restoring a program state from the active thread updates the “active thread” pointer and saving a program state to a non-active thread updates the corresponds pointer in our array of thread pointers. A switch to a thread amounts to assigning the corresponding pointer in our array of thread pointers to the “active thread” pointer.

Next, we look at an example of the application of threads. When global mutable data are modified, there are several possible changes regarding program states. Some of the saved states become unavailable because the sequence of computation that lead to that program state depended on the part of the mutable global data that has been modified. For example, suppose that we have a global mutable object G and a function f0 that does the following: if test(G) is true, then it returns the result of some computation independent of G; otherwise, it returns 0.

```
int
f0() {
  if(test(G)) {
    return computeSomethingIndependentOfG();
  } else {
    return 0;
  }
}
```

Also, suppose that we have a function "flip" which modifies G so that the test(G) flips. We use the function as follows:

```
G = ...; // initialize G so that it makes test(G) true
System.out.println(f0());
flip(); // modify G so that it makes test(G) false
System.out.println(f0());
flip(); // modify G so that it makes test(G) true
System.out.println(f0());
```

It is obvious that at the end of its execution, the third value printed should be the same as the first value printed, and the second value is 0. computeSomethingIndependentOfG is called twice, which is a redundancy. To avoid this redundancy, we can write an incremental version of f0 using threads. Our first try looks like the following:

```
int
f1() {
  if(test(G)) {
    p = computeSomethingIndependentOfG();
    anchored in_thread[0];
    return p;
  } else {
    anchored in_thread[1];
    return 0;
  }
}
```

This function has two anchored program states before its two return statements. The anchored program states tell the runtime to save the program states immediately before the program returns, so that they can be restored later. The two anchored program states are saved in different threads, so that we can switch between them by switching between threads. We use this function as follows:

```
G = ...; // initialize G so that it makes test(G) true
counter = 0;
System.out.println(f1());
flip();
System.out.println(f1());
if( counter < 1 ) {
  counter++;
  flip();
  switch_to_thread(test(G)?0:1);
  backtrack;
}
```

The first two calls to f1 remains the same, the third call to f1 is done by switching to the correct thread and backtrack. This way, the redundancy in f0 is reduced.

But there is still a problem with f1! If we use it in the following way:

```
G = ...; // initialize G so that it makes test(G) true
counter = 0;
System.out.println(f1());
```

```
if( counter < 2 ) {
  counter++;
  flip();
  switch_to_thread(test(G)?0:1);
  backtrack;
}
```

the program does not print the second value at all. To see why, after the call to f1, the first anchored program state is saved. The second anchored program state, however, is not saved because the branch of the if statement which it is in has not been executed. After "flip" and after we switch threads, the saved anchor becomes unavailable, and the backtrack statement will not be able to find any available program state to restore. To solve this problem, we need to take future program states into consideration. We modify f1 as follows:

```
int
f1() {
  if(test(G)) {
    either {
      p = computeSomethingIndependentOfG();
      anchored in_thread[0];
      return p;
    } or in_thread[1] {
      anchored in_thread[1];
      return 0;
    }
  } else {
    either {
      anchored in_thread[1];
      return 0;
    } or in_thread[0] {
      p = computeSomethingIndependentOfG();
      anchored in_thread[0];
      return p;
    }
  }
}
```

To see how this works, when f1 is called, it executes “test(G)” and depending on its result, it returns the correct value. Before it saves the anchored program state, it saves a non-anchor program state in a different thread. Since this program state is in a different thread from the thread where the anchor is saved, it is restored, if and only if we have switched to that thread, if and only if test(G) gives a different result.[1]

6.2 Explicit Global Mutable Data Dependency

In this subsection, we will describe one of the most important features of the STACK EL – explicit global mutable data dependency. Here, “explicit global mutable data dependency” has two dimensions:

1. The STACK EL allows programmers to programmatically mark a program state as being dependent on a piece of global mutable data, so that if the global mutable data has been modified, the program state becomes available or unavailable. On some global mutable data, such as those of the boolean type, the program state can jump back and forth between being available and being unavailable. An example is the producer/consumer problem. If we have a buffer of size 1 to store products, and two program states, one immediately

[1]The actual source code is based on save_state and restore_state which reduces the repeated code.

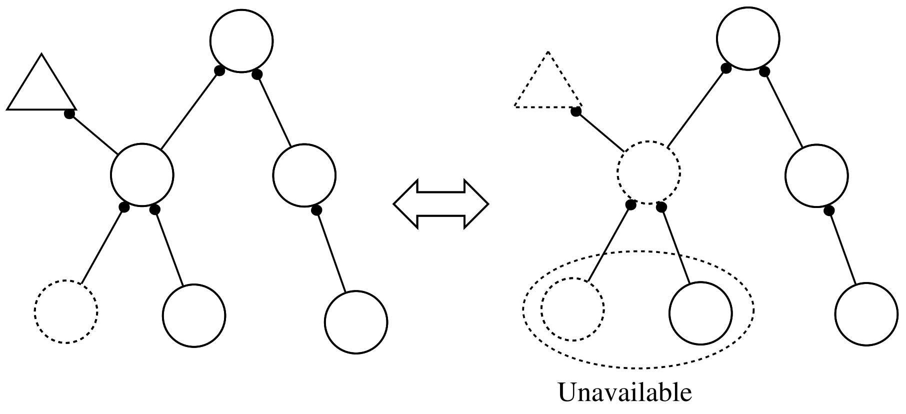

Figure 6.2.5: Explicit Data Dependency

before the producer produces a product, another immediate before the consumer consumes a product, then at any time one of of the program states is available and the other is unavailable. If the buffer is full, then the producer's program state is unavailable and the consumer's program state is available; otherwise, the producer's program state is available and the consumer's program state is unavailable.

2. The STACK EL also allows programmer to programmatically mark a program state as being dependent on another program state. This feature essentially puts program states into a tree[2] and allows programmers to make a whole subtree unavailable by making the root of that subtree unavailable, as illustrated in Figure 6.2.5. In this figure, the circles represent program states and the triangle represents some global mutable data. Available program states are represented by circles with solid line and unavailable ones dotted lines. The line with a dot on one end represents dependency. The global mutable data has two different values, one represented by solid lines, and the other represented by dotted lines. One of the program states has a direct dependency to

[2]Circular dependency is not supported and it is left for programmers to ensure that there is no circular dependency.

the global mutable data. The program state is available when the global data has one value and unavailable when the global data has the other value. When this program state becomes available, program states that depend on it can be either available or unavailable; when it becomes unavailable, all program states that depend on it become unavailable; when it becomes available again, the program states that depend on it restore their original availability.

In our pseudo-code syntax, we introduce the following new constructs.

First, another variant of the the either-or control structure is:

```
either <stmt> or depending_on[<obj ref>] <stmt>
```

where "<obj ref>" is a reference to some mutable global object. We can think of the global object as turning on and off a switch of availability of the program state saved. The program state become available when "<obj ref>" is a value other than the current value. For example, if "<obj ref>" is of boolean type and the current value is true, then the program state becomes available when the value becomes false. An unavailable program state cannot be restored. Accompanying this statement is:

```
update_availability(<obj ref>)
```

which allows us to update the availability of program states depending on the "<obj ref>". And

```
dependency(<obj ref>)
```

which declares that all saved program states in the future will depend on "<obj ref>". The program state remains available when "<obj ref>" remains the same value. For example, if "<obj ref>" is of boolean type and the current value is false, then the program state remains available only when the value remains false. The "dependency" statement creates a "sticky" dependency in that once the statement is encountered, all future saved program states are dependent on it; but the "Sticky" dependency is canceled when the program backtracks to a saved program state that does not depend on "<obj ref>". However, any state that depends on "<obj ref>" still remains dependent on "<obj ref>".

We will see next how these constructs are implemented. We first extend the ProgramState struct from the previous subsection as follows:

```
struct ProgramState {
  void *programPointer;
  StackFrame *stackFrame;
  ProgramState* prevInThread;
  bool anchor;
  int threadId;
  FLAG_TYPE availability;
  ProgramState *super;
  ProgramState *child;
  ProgramState *prev;
  ProgramState *next;
  ProgramState *directDataDependencyNext;
  ...
}
```

As shown in Figure 6.2.5, the dependencies between program states are stored in a tree. We call this kind of trees "indirect data dependency trees." There can be multiple indirect data dependency trees, forming an indirect data dependency forest. Each tree is implemented as a "child-sibling" tree. "child" points to the child node and "next" points to the sibling node. In addition, "super" is the reverse pointer of "child" and "prev" is the reverse pointer of "next". "availability" is a fixed length bit vector type that stores the availability indicator of the current node. There are separate bits to indicate whether this node is disabled by a direct data dependency on global mutable data and whether it is disabled by dependency on another program state. “directDataDependencyNext” is used for a different linked list which we explain next.

To support direct data dependency on global mutable data, we need an extra data structure for each piece of global mutable data which a program state can depend on. The data structure contains pointers to a ProgramState object:

```
struct DirectDataDependency {
  ProgramState *dependentNodesHead;
  ProgramState *dependentNodesTail;
}
```

This structure, together with “directDataDependencyNext” in the ProgramState struct, forms a linked list. For each piece of global mutable data, this linked list stores pointers to all program states that directly depend on this piece of global mutable data.

When saving a state with

```
depending_on[<obj ref>]
```

the saved state is added to the linked list corresponding to the piece of global mutable data referenced in this clause.

There is always an “active dependency state” pointer which points to a program state which a newly saved state depends on. When saving a state, if the “active dependency state” pointer is not null, then the save state is added to the indirect data dependency forest, as a child node of the current active dependency state; otherwise it is added as a root.

Normal SAVE_STATEs do not update the “active dependency state” pointer, To update the “active dependency state” pointer, we use the statement

```
dependency[<obj ref>]
```

It creates a ProgramState object, without associating it with any thread, and updates the "active dependency state" pointer to point to the newly generated ProgramState object.

Finally,

```
update_availability(<obj ref>)
```

is implemented as one function for each piece of global mutable data. The function maps the state of the piece of global mutable data to the availability of the program state that directly depends on it.

Explicit data dependency can be used together with threads. Program states in one thread may depend on program states in another thread.

Next, we look at an example of how to apply explicit data dependency.

We follow the example from the previous subsection and see how it can be implemented using explicit data dependency instead of threads. Recall that we have a function

```
int
f0() {
  if(test(G)) {
    return computeSomethingIndependentOfG();
  } else {
    return 0;
  }
}
```

which can be used as follows:

```
G = ...; // initialize G so that it makes test(G) true
System.out.println(f0());
flip(); // modify G so that it makes test(G) false
System.out.println(f0());
```

```
flip(); // modify G so that it makes test(G) true
System.out.println(f0());
```

It is obvious that at the end of its execution, the third value printed should be the same as the first value printed, and the second value is 0. computeSomethingIndependentOfG is called twice which is a redundancy.

To avoid this redundancy, we can write an incremental version of f0, this time using explicit data dependency:

```
int
f1() {
  if(test(G)) {
    p = computeSomethingIndependentOfG();
    anchored depending_on[G];
    return p;
  } else {
    anchored depending_on[G];
    return 0;
  }
}
```

We use this function as follows:

```
G = ...; // initialize G so that it makes test(G) true
counter = 0;
System.out.println(f1());
flip();
System.out.println(f1());
if( counter < 1 ) {
  counter++;
  flip();
  backtrack;
}
```

Now, we do not need to explicitly call "switch_to_thread." However, the burden has been shifted to the "flip" function. We need to modify "flip" to call "update_availability".

```
void
flip() {
  ...
  update_availability(G);
}
```

Similar to the previous example, if we use it in the following way:

```
G = ...; // initialize G so that it makes test(G) true
counter = 0;
System.out.println(f1());
if( counter < 2 ) {
  counter++;
  flip();
  backtrack;
}
```

then we need to take future program states into consideration:

```
int
f1() {
  if(test(G)) {
    either {
      p = computeSomethingIndependentOfG();
      anchored depending_on[G];
      return p;
    } or {
      anchored depending_on[G];
      return 0;
    }
  } else {
    either {
      anchored depending_on[G];
      return 0;
    } or {
      p = computeSomethingIndependentOfG();
      anchored depending_on[G];
      return p;
    }
  }
}
```

To see how this works, when f1 is called, it executes “test(G)” and depending on the result, it returns the correct value. Before it saves the anchored program state, it saves a non-anchored program state with an either-or branch. Since this program state is saved before the anchor, it is below the anchor in the “active thread.” Therefore, it is only restored when the anchored program state is unavailable, which means that “test(G)” has been flipped to a different value.[3]

6.2 Static Checking of the STACK EL

It should be clear now that we have implemented a runtime whose behavior vastly differs from that of the system stack. These new constructs do not have the same level of support from the host language compiler as host language constructs do. A naive design would result in an embedded language with minimum static checking, resulting in embedded language programs that are error-prone and hard to debug. STACK EL has been designed to ensure that the embedded language enjoys the some of the basic type safety provided by C with little or no runtime

[3]The source code is based on save_state and restore_state which reduces the repeated code.

overhead. It essentially acts as a translator from the problem of static checking on the embedded language to the problem of static checking on the host language.

The implementation of the static checking for the STACK EL borrowed many of the ideas from Internet forums. Some of the ideas have been used, although in quite different forms, in the implementation of the Boost MPL [18]. Therefore, we are not going to reiterate them in details here. What we want to emphasize is that as far as we know, the STACK EL is the first macro-based embedded language that implements all of the following features:

- It is mainly based on C/C++ macros, no external tools are need. Therefore, it is portable.
- It is an embedded language that provides a programming interface similar to a generic programming language.
- It systematically integrates static checks with the EL syntax with little or no runtime over head, such as:
  1. The syntax is correct, such as the PARAMS-RETURNS-BEGIN-END sequence
  2. EL function parameters usages are correctly typed
  3. EL local variables usages are the correctly typed
  4. The number of argument in an EL function call matches the corresponding EL function declaration.
  5. The types the EL arguments match the corresponding EL function declaration.
  6. The number of values an EL function returns matches the corresponding EL function declaration.

7. The types of values an EL function returns matches the corresponding EL function declaration.

8. The return type of the callee in a tail function call matches the return type of the caller.

As an example of static checking, suppose that we have

```
#define FUN lookup
  PARAMS(
    Trie *, curr,
    Sub *, sub)
  RETURNS()

  BEGIN
  DEFS(Trie *, varNode)
  ...
  END
```

The following

```
VAR(varNode)=VAR(curr)
```

compiles but the following

```
VAR(sub)=VAR(curr)
```

does not. The latter would generate a type error by the (unmodified) g++ compiler:

```
error: cannot convert '__type_lookup_curr␣{aka␣Trie*}' to
'__type_lookup_sub␣{aka␣LinkedStack<Term*>*}' in assignment
```

In fact, if one writes STACK EL programs using an integrated development environment (IDE) such at Eclipse or Netbeans, since the STACK EL is compatible with the auto-completion feature of those IDEs, the IDEs may show a list of suggestions when one types "->" after a "VAR" macro. (Of course, credits for the auto-completion feature should be given to the IDE developers.)

Chapter 7

incOSHL, the Incremental Version

In this chapter, we look at the incremental version of our incOSHL algorithm that we mentioned earlier. The basic idea of the incremental version is:

1. After it fails to find an instance of certain size, it incrementally searches for an instance of a different size.

2. After it finds an instance, it incrementally searches for the next instance.

Here, being incremental means that the computation that has been performed before is not performed again. For example, if a subtree of the global trie has been fully traversed and no instance was found (regardless of size), then that subtree should not be traversed again.

The subtle part here is how to deal with changing global data. For example, even if we have traversed a subtree of the global trie and have not found any instance, if a new term is inserted which happens to be in that subtree, then we need to revisit that subtree when generating a new instance. However, every time we revisit an already visited subtree, we should only revisit the nodes that has been added, removed, or modified.

7.1 Application of Threads

In this section we look at how to incorporate a simple global data dependency to improve the efficiency of our algorithms: program states that depend on the value of "globalSizeLimit." In our modified algorithm, we have multiple future threads; each thread is given a unique integral id indicating the value of "globalSizeLimit" required for states in this thread to become available.

First, we take a look at the modified "lookup" algorithm. We will use a syntax sugar

```
resume in_thread[X]
```

where X is a parameter. It is equivalent to

```
either
  backtrack;
or in_thread[X]
  ; // do nothing
```

This construct is used when a subroutine detects that the generated instances will have a size larger than "globalSizeLimit." It saves the current program state in thread X, backtracks, and resumes when thread X becomes active.

The "lookup" subroutine is modified as follows:

```
(TrieNode, Sub, int)
lookup(int n, TrieNode node, Symbol symbol, Sub sub) {
  if( symbol == null ) {
    return (node, sub, n);
  } else if( symbol.key.isFuncSymbol ) {
    nodeNew = node.lookup(symbol.key);
    if( ! nodeNew.available ) {
      backtrack;
    } else {
      return lookup(n, nodeNew, symbol.next, sub);
    }
  } else { // symbol is a variable
    subterm = sub.lookup(symbol);
    if( subterm == null ) {
      (nodeNew, retterm, nNew) = branch(n, cnode, cnode, 1);
      subNew = sub.push(symbol.key, retterm);
      return lookup(nNew, nodeNew, symbol.next, subNew);
    } else {
      nInput = n + subterm.termSize - 1;
      if( nInput > globalSizeLimit) {
        resume in_thread[nInput];
      }
      (nodeNew, subNew, nNew) =
        lookup(nInput, node, subterm.firstSymbol, sub);
      return lookup(nNew, nodeNew, symbol.next, subNew);
    }
  }
}
```

The only change is highlighted. Comparing with the non-incremental version of this subroutine, this version stores the program state in thread with id “nInput” before backtracking.

The “branch” subroutine, the “lookupInverse” subroutine, and the “branchInverse” subroutine can be modified similarly.

```
(TrieNode, Term, int)
branch(int n, TrieNode start, TrieNode curr, int np) {
  if( numberOfPlaceholders == 0 ) {
    return (curr, curr.subterm(start), n);
  } else {
    return branchTryTargetNodes(n, start, curr, 1);
  }
}


(TrieNode, Term, int)
branchTryTargetNodes(int n, TrieNode start, TrieNode curr, int np) {
  either {
    nInput = n + curr.symbol.arity;
    if( nInput > globalSizeLimit ) {
      resume in_thread[nInput];
    }
    return branch(nInput, start, curr, np + arity - 1));
  } or {
    if(curr.sibling == null) {
      backtrack;
    }
    return branchTryTargetNodes(n, start, curr.sibling, np);
  }
}
```

```
(TrieNode, Sub, int)
lookupInverse(int n, TrieNode node, Symbol symbol, Sub sub) {
  if( symbol == null ) {
    return (node, sub, n);
  } else if( symbol.key.isFuncSymbol ) {
    nodeNew = node.lookUp(symbol.key);
    return lookupInverse(n, nodeNew, symbol.next, sub);
  } else { // symbol is a variable
    subterm = sub.lookup(symbol);
    if( subterm == null ) {
      (nodeNew, retterm, nNew) = branchInverse(n, node, node, 1);
      subNew = sub.push(symbol.key, retterm);
      return lookupInverse(nNew, nodeNew, symbol.next, subNew);
    } else {
      nInput = n + subterm->termSize - 1;
      if( nInput > globalSizeLimit) {
        resume in_thread[nInput];
      }
      (nodeNew, subNew, nNew) =
        lookupInverse(nInput, node, subterm.firstSymbol, sub);
      return lookupInverse(nNew, nodeNew, symbol.next, subNew);
    }
  }
}
```

```
(TrieNode, Term, int)
branchInverse(int n, TermCacheNode currTCN, TrieNode node,
  int nPlaceholders) {
  if( nPlaceholders == 0 ) {
    return (node, currTCN.term, n);
  } else {
    return branchInverseTryFuncSymbols(n, currTCN, node,
      nPlaceholders, 0);
  }
}


(TrieNode, Term, int)
branchInverseTryFuncSymbols(int n, TermCacheNode superTCN,
  TrieNode node, int nPlaceholders, int i) {
  either {
    s = functionSymbols[i];
    newTCN = superTCN.lookup(s);
    nodeNew = node.lookup(s);
    nInput = n + s.arity;
    if(nInput > globalSizeLimi) {
      resume in_thread[nInput];
    }
    return branchInverse(nInput, newTCN, nodeNew,
      nPlaceholders + s.arity - 1);
  } or if(i + 1 != numberOfFunctionSymbols ) {
    return branchInverseTryFuncSymbols(n, superTCN, node,
      nPlaceholders, i + 1);
  }
}
```

The “traverse” subroutine and “traverseSubtrees” subroutine remain unchanged.

The mainLoop can now be modified to take advantage of threads:

```
boolean
mainLoop() {
  for(;;) {
    truncate;
    for(i = 1; i <= maxThreadId; i ++) {
      either
        ; // do nothing
      or anchored in_thread[i]; {
        if(saturated()) {
          return true;
        }
        globalSizeLimit ++;
        switch_to_thread(globalSizeLimit);
        backtrack;
      }
    }
    globalSizeLimit = 1;
    switch_to_thread(1);
    (noderet, subret) = traverse(clauseTreeRoot, new Sub());
    if( ! updateModel(noderet, subret) ) {
      return false;
    }
  }
}
```

Compared with the non-incremental version, this algorithm does not have the loop that increments "globalSizeLimit." Instead, it has an initial loop that saves in each thread an anchored program state that handles the failure condition of the traverse subroutine. Each iteration of the outer loop either generates an instance or returns. Inside the loop, the subroutine first the truncates the stack; then, it sets up the anchors; after that, it sets "globalSizeLimit" and switches to the thread with id 1; after that, it calls the "traverse" subroutine to search for a clause instance. If an instance is found, then it updates the model and either returns if the update fails, or goes to the next iteration of the outer loop; if an instance is not found, then it backtracks, increments "globalSizeLimit," switches to a new thread, and tries again.

Note that we still need the truncate statement but on a higher level, since the updateModel subroutine modifies other global mutable data that we have not taken into consideration so far.

7.2 Application of Explicit Global Mutable Data Dependency

Now let us take a look at how we can make use of explicit global mutable data dependency to improve the efficiency of our algorithm. In the previous section, we have already seen how we can reuse computation when the value of "globalSizeLimit" increases. Now, we want to reuse computation when the trie storing all literals in the current model is modified. This modification essentially makes our theorem prover scan each path of the search tree only once, as opposed to many times in the versions we have seen so far. Every new instance is generated incrementally.

First, the mainLoop now looks like:

```
boolean
mainLoop() {
  globalSizeLimit = 1;
  for(i = 1; i <= maxThreadId; i ++) {
    either
      ; // do nothing
    or anchored in_thread[i]; {
      if(saturated()) {
        return true;
      }
      globalSizeLimit ++;
      switch_to_thread(globalSizeLimit);
      backtrack;
    }
  }
  switch_to_thread(1);
  (noderet , subret) = traverse(clauseTreeRoot , new Sub());
  if( ! updateModel(noderet , subret) ) {
    return false;
  }
  switch_to_thread(1);
  backtrack;
}
```

Compared with the previous version, this algorithm does not have the outer loop. We have deleted the truncate statement and added two statements at the end: switch_to_thread(1) and backtrack. It still has the initial loop that saves in each thread an anchored program state that handles the failure condition of the traverse subroutine. The subroutine first sets up the anchors; then, it sets globalSizeLimit and switches to the thread with id 1; after that, it calls the "traverse" subroutine to search for a clause instance. If an instance is found, then it updates the model and either returns if the update fails, or switches back to thread with id 1, backtracks, and searches for a clause instance that contradicts the new model; if an instance is not found, then it backtracks, increments “globalSizeLimit,” switches to a new thread, and tries again.

To see how this works, first notice that the “updateModel” subroutine modifies the global state so that the availabilities of program states may change during a call to this subroutine. Since we are not truncating the threads any more, we can backtrack at the end of mainLoop. When doing the backtracking, we will go back to a previous saved state that is available after the modification of our model. As long as the dependencies are correctly encoded, our incremental version works to the same effect as our non-incremental version.

Next, let us take a look at how the trie can be modified. There are two ways to modify our trie. One is to insert a literal. Those literals are in the set Add_k. The other is to delete a literal. Those literals are in the set Del_k. Logically, when adding a literal, some nodes, internal or leaf, will be added to the trie. If any of these nodes already has a TrieNode object created but its “available” field is set of false, then adding the node to the trie simply amounts to flipping the “available” field. When deleting a literal, some nodes, internal or leaf, will be deleted from the trie. Similarly, deleting a node simply amounts to flipping the “available” field.

Now let us take a brief moment to think about examples of what data dependencies we need to take into consideration when modifying out subroutines.

1. Suppose that we have a literal L, if (inverse) lookup of L fails in the current trie, it does not mean (inverse) lookup of L will fail in all future tries. We need to save the program state where (inverse) lookup fails, and come back later when the trie changes. This a case of dependency on future objects that

have not been created yet. For example, suppose that we have a literal $f(a, b)$ which is not in our trie. If we run the "lookup" subroutine we listed in the previous section to look for an instance of our literal, which can only be itself, in our trie, the "lookup" subroutine will simply backtrack out, without saving any state. Even if $f(a, b)$ is added into our trie in the future, the "lookup" subroutine in the previous section will not revisit it anymore unless it starts afresh.

2. Suppose that we have a clause with two negative literals L and N. If I_0 make all negative literals true, then in order for an instance of the clause to be false, we need to match both $\overline{L}$ and $\overline{N}$ to literals in the current trie. This is done by calling the "lookup" subroutine, twice. After the first call, it will generate a substitution θ, which we will use as the initial substitution for the second call. There is a dependency right here. Any action that will be taken in the second call depends on the data that the literal $\theta(\overline{L})$ is in the trie. Naturally, we want to make any state generated during the second call depend on the global object which is the leaf node in the trie that represents $\theta(\overline{L})$.

3. Following the previous example, if we have also found an instance in the trie for $\overline{N}$, then we have another dependency from any future actions after the second call to that instance in the trie. In particular, when the second call to "lookup" returns, we have a new substitution θ' and an instance of the input clauses which contradicts the current model. Whether this instance of the input clauses contradicts a model partly depends on whether $\theta'(\overline{N})$ is the trie that represents that model.

To incorporate these dependencies, we modify the "traverse" subroutine. We will use a syntax sugar

```
resume depending_on[X]
```

where X is a parameter. It is equivalent to

```
either
  backtrack;
or depending_on[X]
  ; // do nothing
```

The idea is that we save the current program state but make it unavailable if X has not been changed, backtrack, and resume when X has been changed.

The “traverse” subroutine looks like:

```
(ClauseTreeNode, Sub)
traverse(ClauseTreeNode node, Sub subInit) {
  if(node.literal == null) {
    retsub = subInit;
  } else if(node.literal.isPositive) {
    (retnode, retsub, _) = lookupInverse(node.literal.atom.termSize,
      trieRoot, node.literal.atom.firstSymbol, subInit);
    if( retnode.available ) {
      resume depending_on[retnode.available];
    }
  } else {
    (retnode, retsub, _) = lookup(node.literal.atom.termSize,
      trieRoot, node.literal.atom.firstSymbol, subInit);
  }
  dependency(retnode.available);
  if(node.isLeaf) {
    anchored;
    return (node, retsub);
  } else {
    return traverseSubtrees(node, retsub, 0);
  }
}
```

The resume statement encodes 1 on Page 191. The dependency statement encodes 2 on Page 192. The anchor statement encodes 3 on Page 192.

To incorporate 3 on Page 192, we also need to modify the "lookup" subroutine.

```
(TrieNode, Sub, int)
lookup(int n, TrieNode node, Symbol symbol, Sub sub) {
  if( symbol == null ) {
    return (node, sub, n);
  } else if( symbol.key.isFuncSymbol ) {
    nodeNew = node.lookup(symbol.key);
    if( ! nodeNew.available ) {
      resume depending_on[newNode.available];
    }
    return lookup(n, nodeNew, symbol.next, sub);
  } else { // symbol is a variable
    subterm = sub.lookup(symbol);
    if( subterm == null ) {
      (nodeNew, retterm, nNew) = branch(n, cnode, cnode, 1);
      subNew = sub.push(symbol.key, retterm);
      return lookup(nNew, nodeNew, symbol.next, subNew);
    } else {
      nInput = n + subterm.termSize - 1;
      if( nInput > globalSizeLimit) {
        resume in_thread[nInput];
      }
      (nodeNew, subNew, nNew) =
        lookup(nInput, node, subterm.firstSymbol, sub);
      return lookup(nNew, nodeNew, symbol.next, subNew);
    }
  }
}
```

The highlighted code saves the current state for possible future exploration. We need also to make a similar modification to other subroutines to incorporate 3 on Page 192. The "branch" subroutine is modified as follows:

```
(TrieNode, Term, int)
branch(int n, TrieNode start, TrieNode curr, int np) {
  if( numberOfPlaceholders == 0 ) {
    return (curr, curr.subterm(start), n);
  } else {
    return branchTryTargetNodes(n, start, curr, 1);
  }
}


(TrieNode, Term, int)
branchTryTargetNodes(int n, TrieNode start, TrieNode curr, int np) {
  either {
    nInput = n + curr.symbol.arity;
    if( nInput > globalSizeLimit ) {
      resume in_thread[nInput];
    }
    return branch(nInput, start, curr, np + arity - 1));
  } or {
    if(curr.sibling == null) {
      resume depending_on[curr.sibling];
    }
    return branchTryTargetNodes(n, start, curr.sibling, np);
  }
}
```

The "lookupInverse" and the "branchInverse" subroutine remain unchanged. The

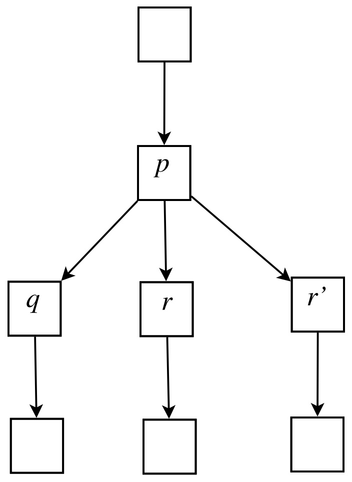

Figure 7.3.1: A Clause Tree

"updateModel" subroutine does not have to be modified, either, but the "insert" and "remove" methods in the "TrieNode" class have to be modified so that when adding or deleting a trie node, the availability of the program states that directly or indirectly depend on the trie node will be updated, using the "update_availability" statement. We leave the details of this modification to the source code.

7.3 Optimality of the Incremental Version of incOSHL

In this section, we establish the optimality of the incremental version of incOSHL. Before giving a definition of optimality, we first look at an example that is considered non-optimal. Take the non-incremental version of incOSHL for example. Suppose that

1. $p <_{sl} q <_{sl} r <_{sl} r'$
2. the input set S of clauses is $\{\{p, q\}, \{p, r\}, \{p, r'\}\}$,
3. the initial interpretation I_0 makes all negative literals true,
4. we are at iteration k,

5. $T_k = \{\{p, q\}\}$, and

6. $M_k = \{q\}$.

The clause tree constructed from S looks like that in Figure 7.3.1. It is easy to see that one instance that contradicts the current model is $\{p, r\}$. If we simulate the non-incremental version of incOSHL from Chapter 5, the list of steps that are needed for generating $\{p, r\}$ are

1. run "lookupInverse" on p, success

2. run "lookupInverse" on q, failure, backtrack

3. run "lookupInverse" on r, success, return $\{p, r\}$

Now, we generate T_{k+1} and M_{k+1} where

1. $T_{k+1} = \{\{p, q\}, \{p, r\}\}$ and

2. $M_{k+1} = \{q, r\}$

Now, we try to generate another instance that contradicts the current model (I_0, M_{k+1}). This instance can only be $\{p, r'\}$. If we simulate the non-incremental version of incOSHL from Chapter 5 again, the list of steps that are needed for generating $\{p, r'\}$ are

1. run "lookupInverse" on p, success

2. run "lookupInverse" on q, failure, backtrack

3. run "lookupInverse" on r, failure, backtrack

4. run "lookupInverse" on r', success, return $\{p, r'\}$

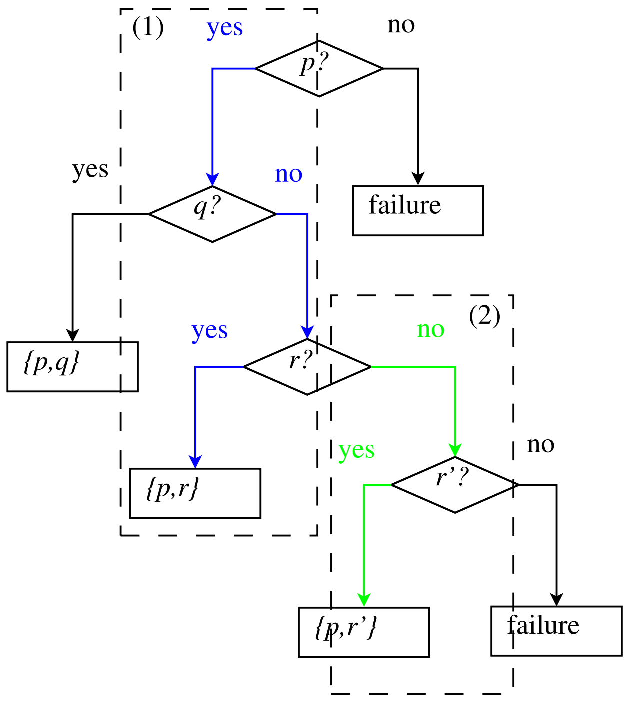

Figure 7.3.2: A Decision Tree

Apparently, Step 1 and Step 2 are simply repeating Step 1 and Step 2 of the previous iteration. This is redundant computation. We can see this by looking at the decision tree in Figure 7.3.2. In the kth iteration, we have already made the path of decision in dashed box (1). In the $k+1$ iteration, we do not need to remake the first two decisions in that path again, since adding r to M_{k+1} does not affect those decisions that we have made; however, we do need to remake the third decision in that path, and switch to the new decision in dashed box (2), since adding r to M_{k+1} does affect this decision.

Our incremental version of incOSHL, to the contrary, does not have any redundancy of this kind. If in the kth iteration, we made the decision in dashed box (1), then in the $k+1$st iteration, we only need to make the decision in dashed box (2). In general, the incremental version of incOSHL has the property that once an edge in the decision tree has been visited, the "traverse" subroutine and any subroutine that it calls will not revisit this edge anymore.

In the following discussion, we use the following minimal untyped formal language: We assume that f generates all function names, $v, v_1, v_2, \ldots$ generate all variables, tid generates all thread ids, and ref generates all object references.

$prog$	$::=$	$fdef_1\ \ldots\ fdef_n\ stmt$	program
$fdef$	$::=$	$f(v_1, v_2, \ldots, v_n)\ stmt$	function definition
$stmt$	$::=$	$v_1 = v_2;$	assignment
	$\mid$	$v = f(v_1, v_2, \ldots, v_n);$	function call
	$\mid$	if(v) $stmt_1$ else $stmt_2$	conditional
	$\mid$	either $stmt_1$ or $mod_1\ \ldots\ mod_n\ stmt_2$	either-or
	$\mid$	anchored;	anchor
	$\mid$	dependency;	dependency
	$\mid$	backtrack;	backtrack
	$\mid$	switch_to_thread[id];	switch thread
	$\mid$	update_availability[ref];	update availability
	$\mid$	return v;	return
	$\mid$	$\{stmt_1\ \ldots\ stmt_n\}$	block
mod	$::=$	in_thread[id]	thread
	$\mid$	depending_on[ref]	dependency
	$\mid$	anchored	anchored

This formal language is sufficient given that we implement all constants and operators as functions.

Now we give a formal definition of a decision tree. We represent the decision tree as nested tuples. A tuple (x, y, z) denotes a tree node x with left subtree y and right subtree z. For example, $(y, success, failure)$ denotes a decision tree node y with left subtree $success$ and right subtree $failure$. Similarly,

$(y,(y,success,failure),failure)$ denotes a decision tree node y with left subtree $(y,success,failure)$ and right subtree $failure$. We say there is an edge from every node to any of its subtrees. Our algorithm for generating decision trees uses the following tree concatenation operator $\oplus$: Given two decision trees T_1 and T_2, $T_1 \oplus T_2$ is defined as replacing all *success* in T_1 by T_2. For example, if

$$T_1 = (y, success, failure)$$

and

$$T_2 = (x, success, failure)$$

then

$$T_1 \oplus T_2 = (y, (x, success, failure), failure)$$

Definition 102. Given a sequence Q of statements, the decision tree $\mathcal{D}(Q)$ is defined as follows:

1. If Q is of the form backtrack; Q', where Q' is a sequence of statements, then $\mathcal{D}(Q) = failure$.

2. If Q is of the form $x = f(y); Q'$, where x and y are variable, f is a function defined by statement P, and Q is a sequence of statements, then $\mathcal{D}(Q) = \mathcal{D}(P) \oplus \mathcal{D}(Q')$.

3. If Q is of the form return $e; Q'$, where e is an expression without function calls in it and Q' is a sequence of statement, then $\mathcal{D}(Q) = success$.

4. If Q is of the form if(e) s else $t; Q'$, where e is a boolean expression without

function calls in it, s and t are statements, and Q' is a sequence of statements, then $\mathcal{D}(Q) = (e, \mathcal{D}(s\ Q'), \mathcal{D}(t\ Q'))$.

5. If Q is of the form either s or $mod_1\ \ldots\ mod_n\ t; Q'$, where s and t are statements and Q' is a sequence of statements, then $\mathcal{D}(Q) = (e, \mathcal{D}(s\ Q'), \mathcal{D}(t\ Q'))$.

6. If Q is of the form $\{s_1 \ldots s_n\}Q'$, where $s_1, \ldots, s_n$ are statements, Q' is a sequence of statements, then $\mathcal{D}(Q) = \mathcal{D}(s_1 \ldots s_n Q')$.

7. If Q is of the form sQ', where s is a statement, Q' is a sequence of statements, and none of the above applies, then $\mathcal{D}(Q) = \mathcal{D}(Q')$.

For example, if we have the following pseudo-code

```
boolean
f(int x) {
  y = x + 1;
  if( y > 0 )
    return true;
  else
    backtrack
}
```

then we can convert it to our minimal language:

```
f(x) {
  zero = 0;
  one = 1;
  y = +(x,one);
  e = >(y,zero);
  if( e ) {
    t = true;
    return t;
  }
```

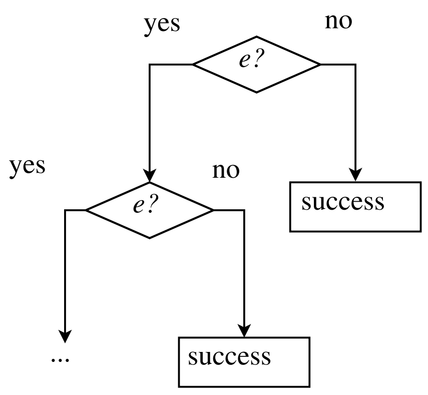

Figure 7.3.3: Decision Tree of "factorial"

```
  else
    backtrack;
}
```

The decision tree of the sequence of statements in "f" looks like $(e, success, failure)$.

For another example, if we have the following function written in our pseudo-code:

```
int
factorial(int x) {
  if( x > 0 ) {
    return x * factorial(x - 1);
  else
    return 1;
}
```

then we can convert it to the following in our minimal language:

```
factorial(x) {
  zero = 0;
  one = 1;
  e = >(x,zero);
  if( e ) {
    y = -(x,one);
    z = factorial(y);
    f = *(z,x);
    return f;
  } else {
    return one;
  }
}
```

The decision tree of the sequence of statement in “factorial” is an infinite tree that looks like $(e, (e, ..., success), success)$, as shown in Figure 7.3.3. It is clear that the decision tree includes all possible paths for all possible inputs to this function.

Now, if we look at our incremental version of the “mainLoop” subroutine from Section 7.2, then we can see that the traverse subroutine is only called once. Therefore we have the following formal definition of optimality:

Definition 103. The “mainLoop” is optimal if no edge in the decision tree

$$\mathcal{D}(\text{traverse}(\text{clauseTreeRoot},\textsf{new}\ \text{Sub}()))$$

is visited twice.

We have already seen an example where the non-incremental version of incOSHL is not optimal. Now we prove that

Theorem 104. *The incremental version of incOSHL is optimal.*

Proof. First, observe that mainLoop only calls the traverse subroutine once. Second, we observe that all anchored program states lead to returning to the mainLoop. Therefore there are no edges under any anchored program state, i.e., backtracking to anchored program states will not cause double visit of an edge in the decision tree. □

The remaining question is what does Theorem 104 say exactly in terms of instance generation? To answer this question we need to look at our decision tree. Each path in our decision tree either leads to failure or a unique Sub object and ClauseTreeNode object pair. The significance of Theorem 104 is that using it we can show that

Theorem 105. *mainLoop generates every Sub object and ClauseTreeNode object pair at most once.*

Proof. This can be proved by examining every if statement and every either-or structure.

There are two kinds of if statements. The first kind of if statement has a condition whose value does not depend on external data. Only one branch will ever be executed in this kind of if statement. Therefore, it has the same effect as the branch that actually get executed. The second kind of if statement has a condition whose value depends on external data, but encloses only one resume statement. By the definition of decision trees, this kind of if statements only produce one branch that is not failure. Therefore, the if statement has the same effect as the non-failure branch.

All the either-or statements have two branches that choose either different branches in the clause tree, or different branches in the trie. If they choose different branches in the clause tree, then the eventually returned ClauseTreeNode object will be different, if they choose different branch in the trie, then the eventually returned Sub object will be different.

By Theorem 104, mainLoop only generate any Sub object and ClauseTreeNode object pair once. □

The remaining question now is: how significant is the time spent on the "traverse" subroutine versus the time spent on the rest of the prover, mainly the "updateModel" subroutine. This will be one of the topics of the next chapter where we present test results of our theorem prover.

Before concluding this section, we take a look at STACK EL from a different prospective. In a broader sense, the incremental version of incOSHL can be thought of as implementing a multi-level cache that has the following properties:

1. Caching: This is the basic point of a cache. If two sequences of computational steps, even if they lead to different results, share a common initial sequence of computational steps, then the result of the shared initial sequence is cached when one sequence of computational steps is performed; when the second sequence of computational steps is performed, the result of the shared initial sequence is retrieved efficiently without performing that sequence again. Theorem 104 shows that this is true. It essentially guarantees that each computational step is only computed once. The caches, in essence, are the STACK EL stack.

2. Lazy evaluation: Computational steps are executed in a lazy manner. When there is a “don’t know” nondeterminism, only one branch is tried. The second branch is tried only if the first branch does not generate a solution. This is guaranteed by the either-or control structure.

3. Updatability: Global mutable data can be updated incrementally. This can be achieved either using threads or explicit data dependency. The STACK EL, when combined with proper programming, guarantees that the available saved program states are always consistent with the current state of global mutable data.

4. Composability: Because the caches are the STACK EL stack, different components of the cache can be composed in any way a normal program can be

composed. Cache components can be nested like functions are nested. For example, we can considered the caches for instances of literals as nested caches of the cache for instances of the clause. This is naturally done by simply calling the subroutine for generating literal instances within the subroutine for generating clause instance. Similarly, caches can also be put side by side by an either-or structure or an if statement.

5. Customizability: Like composability, the caches can be customized in any way a program can be customized. This makes the caches extremely flexible.

6. Static Checking and Optimization: Since STACK EL is implemented as macros and the macro are expanded during the compile time, the static checking mechanism also works during compile time. The STACK EL also makes heavy use of constant expressions, taking advantage of the optimization of the underlying compiler.

7.4 Other Features

We mention in this chapter some other features of our theorem prover. One of the features is randomization of the input clauses. Since the result of the prover is highly dependent on certain key orders, randomizing these orders will allow us to avoid being fixed on a particular configuration. The orders that can be randomized are:

1. The lexical order on function or predicate symbols $\leq_l$.

2. The order of literals in a clause, when inserting the clause into the clause tree.

3. The order in which clauses are inserted into the clause tree.

Another feature is super symbols. A super symbol is a combination of several adjacent function symbols in a term. For example, if we have term $g(a, f(X))$, then we combine g, a, and f into a super symbol. Using super symbols reduces the number of recursions in "lookup" and "lookupInverse" in two ways:

1. It reduces the number of symbols

2. It ensures that a function, predicate, or super symbol is alway either followed by a variable symbol or the end of a term. This way we can combine the lookup/lookupInverse of the non-variable symbol with the lookup/lookupInverse of a variable symbol into one function call.

When the super symbol feature is enabled, the prover tries combine as many function symbols into super symbols as possible. Super symbol works purely on the implementation level, and it does not affect the semantics, in particular the term order, of the proof strategy.

A third feature is relevance [27]. The relevance of a clause instance is defined as the number of inference steps used to generate the instance. Our theorem prover implemented an optional strategy of favoring instances with lower relevance.

A fourth feature is sorting of clause trees. Our prover allows each node to precompute and store the lower bound of any instances generated by a clause under it and sort the subtrees of a node by their lower bounds in ascending order. This way the "traverse" subroutine can stop searching when it sees a subtree with a lower bound that exceeds the current "globalSizeLimit" because any subtrees after it will have a higher or equal lower bound.

Finally, for our debugging and profiling needs, our STACK EL and theorem prover come with functions that generate logs that can be used for offline debugging, functions that provide online verification of intermediate results and functions that generate profiling data.

Chapter 8

Performance Study

We selected from the latest TPTP [43] library (v5.3.0) all problems that are unsatisfiable, in clausal normal form, and without equality, and used this subset for our tests. This subset includes 1528 problems with varying difficulty ratings and across multiple problem categories. Their distribution is given in Table 8.1. The first column is the name of the TPTP category; the second column is the number of problems in our subset in that category; the third column is the average rating of the problems in our subset in that category. The TPTP technical report gave a detailed description of the rating: "... gives the difficulty of the problem, measured relative to state-of-the-art ATP systems ... The rating is a real number in the range 0.0 to 1.0, where 0.0 means that all state-of-the-art ATP systems can solve the problem (i.e., the problem is easy), and 1.0 means no state-of-the-art ATP system can solve the problem (i.e., the problem is hard)." [44]

Each TPTP category contains a certain type of problems. Among the twenty five categories in our problem set, ALG stands for General Algebra, ANA stands for Analysis, CAT stands for Category Theory, COL stands for combinatory logic, COM stands for Computing Theory, FLD stands for Fields (in Algebra), GEO stands for Geometry, GRA stands for graph theory, GRP stands for Groups (in Algebra), HWV stands for Hardware Verification, KRS stands for Knowledge Representation, LAT stands for Lattices (in Algebra), LCL stands for Logic Calculi, MGT stands

TPTP Category	Total #	Avg. TPTP Rating
ALG	1	0.0000
ANA	16	0.4031
CAT	1	0.0000
COL	21	0.0671
COM	8	0.2900
FLD	161	0.4317
GEO	3	0.4067
GRA	1	0.0000
GRP	21	0.0838
HWV	11	0.0273
KRS	9	0.0000
LAT	11	0.0164
LCL	285	0.3460
MGT	22	0.0273
MSC	18	0.2422
NLP	7	0.0000
NUM	19	0.0937
PLA	43	0.5086
PUZ	60	0.1747
RNG	8	0.2500
ROB	1	1.0000
SET	45	0.0969
SWV	143	0.2608
SYN	608	0.0883
TOP	5	0.0600

Table 8.1: Problem Distribution

for Management, MSC stands for Miscellaneous, NLP stands for Natural Language Processing, NUM stands for Number Theory, PLA stands for Planning, PUZ stands for Puzzles, RNG stands for Rings (in Algebra), ROB stands for Robbins Algebra, SET stands for set theory, SWV stands for Software Verification, SYN stands for Syntactic, and TOP stands for Topology. [3]

We also selected from the latest TPTP [43] library all problems that are satisfiable, in clausal normal form, and without equality, which includes 415 problems, and used this subset to verify the correctness of incOSHL. incOSHL didn't generate any false proof on this problem set.

We ran the test on variations of the following default configuration:

1. The system is UNC's KillDevil Cluster [1]:

   (a) Hardware:

      i. 119 Dell C6100 servers, each with 2.93 GHz Intel X5670 and 48 GB main memory.

      ii. 17 Dell C6100 servers, each with 2.93 GHz Intel X5670 and 96 GB main memory.

      iii. 32 Dell C6100 servers, each with 2.93 GHz Intel X5670 and 48 GB main memory.

      iv. An additional 2 Dell R910 servers, each with 2.00 Ghz Intel X7550 and 1 TB main memory are used to run certain problems with higher memory limit.

   (b) Software:

      i. The operating system running on the cluster is RHEL 5.6 (Tikanga)

      ii. The job and resource management is handled by LSF

      iii. The complier used is g++ (GCC) 4.1.2 20080704 (Red Hat 4.1.2-52)

2. For all jobs, we set a system resource limit to the following unless otherwise stated:

   (a) LSF memory limit is set to 40G. If the memory usage of a job exceeds hard memory limit, then the LSF will kill this job.

   (b) Prover memory limit is set to 20G. This is a parameter passed into all provers.

(c) Hard time limit is set to 300s. If a job runs more than this amount of time, then the job is killed by the GNU coreutils's "timeout" command.

(d) Prover time limit is set to 240s. This is a parameter passed into all provers.

3. incOSHL settings:

   (a) Type inference: by default it is turned on.

   (b) Relevance: by default it is turned off.

   (c) The initial model: by default it makes all negative literals true.

   (d) Profiling is turned off. This means only basic statistics are printed.

8.1 Inference Rate

As we stated in the introduction, one of the main goals of this version of incOSHL is to improve the inference rate over previous implementation of OHSL. Let t_i be the time spent on the ith problem, and n_i be the number of clause instances generated. Let N be the number of problems. The average inference rate is calculated as

$$\frac{\sum_{i=1}^{N} n_i}{\sum_{i=1}^{N} t_i} \tag{8.1.1}$$

The reason we do not calculate the average inference rate as

$$\frac{\sum_{i=1}^{N} \frac{n_i}{t_i}}{N} \tag{8.1.2}$$

is to prevent some problems to dominate the average inference rate. For example, suppose that we have an easy problem that runs for only 0.001 seconds and generated 10 clause instances and a hard problem that runs for 240 seconds and generated

TPTP Category	Total #	Avg. Inf. Rate (inst./s)
ALG	1	8000.0
ANA	16	637.1
CAT	1	550.0
COL	21	350.6
COM	8	8054.8
FLD	161	819.5
GEO	3	12902.2
GRA	1	1600.0
GRP	21	2191.2
HWV	11	18128.6
KRS	9	2800.0
LAT	11	3950.0
LCL	285	18098.8
MGT	22	2131.3
MSC	18	31061.7
NLP	7	88312.5
NUM	19	123.4
PLA	43	3182.3
PUZ	60	6167.9
RNG	8	153.9
ROB	1	31.8
SET	45	61645.1
SWV	143	134481.6
SYN	608	10697.4
TOP	5	725.9
Overall	1528	36043.8

Table 8.2: Inference Rate

10 clause instances. If we use (8.1.2), then the average inference rate is 5.000×10^3 instances per second. If we use (8.1.1), then the inference rate is 8.333×10^{-2}, which more accurately reflects the situation.

Table 8.2 shows the inference rate of incOSHL in each TPTP category and the overall inference rate on all categories. The first column shows the TPTP category. The second column shows the total number of problems in our test set in that category. The third column shows the average inference rate in that category. The inference rate is rounded down to one decimal place. We reuse the first column to

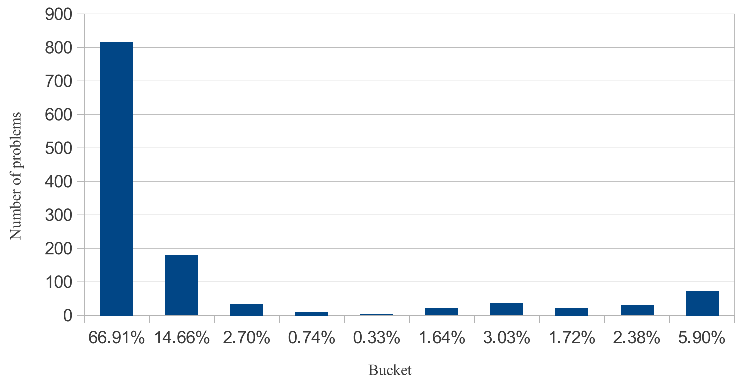

Figure 8.2.1: "updateModel" Time mt_i vs total time t_i. The problems in the kth bucket have the property $\frac{k-1}{10} \leq \frac{mt_i}{t_i} < \frac{k}{10}$.

also include the overall inference rate. Some of the categories have a low average inference rate. Future research may include looking into what caused these categories to have low inference rate.

8.2 "traverse" versus "updateModel"

One question that we raised in the previous chapter was: in practice, how significant is the time spent the "traverse" versus the time spent on the "updateModel" subroutine[1]. If the "updateModel" subroutine takes a small portion of the time, then Theorem 104 is significant, since it shows that "traverse" is not wasting time by performing repeated search. One the other hand, if the "updateModel" subroutine takes a large portion of the time, then it could mean that the overhead for maintaining global mutable date dependency is large and may affect the efficiency of the theorem prover.

On our problem set, we ran incOSHL in the default configuration except that model profiling is turn on, which allows us to profile the total time the prover spend

[1] We only count the "updateModel" time because other overhead is too small to be significant.

on updating models. The statistics of the results are shown in Figure 8.2.1. If we denote the total prover time of problem i by t_i, and the total "updateModel" time of problem i by mt_i, then we want to see whether mt_i/t_i is low on a majority of the problems. To see this, we divided the problems into 10 buckets by their "updateModel" time. We excluded problems whose t_i is 0 [2] (234 in total) and problems whose test run gets killed by the LSF system either for exceeding the hard time limit or LSF memory limit (67 in total). The problems in the kth bucket have the property

$$\frac{k-1}{10} \leq \frac{mt_i}{t_i} < \frac{k}{10}$$

The chart shows that a majority (66.91%) of our problems has a mt_i/t_i ratio of less than 0.1. For the kth bucket, where $k > 1$, the lower bound of average mt_i/t_i ratio is $k/10$. The average mt_i/t_i ratio in the first bucket is 8.701×10^{-3}. This means for a majority of the problems, the time spent on "traverse" dominates the time spent on "updateModel".

8.3 Incremental versus Nonincremental

This section shows how the incremental version of incOSHL compares with the nonincremental version of incOSHL. We implemented both the incremental version and nonincremental version of our algorithm. The only difference between the incremental version and the nonincremental version of our implementation is that the nonincremental version does not use features like threads and explicit data dependency on global mutable data. The nonincremental version still saves program states and backtracks, but limits the program states to only those that can be used when the global mutable data have not been modified. We wrote profiling code to

[2]run time is within the lowest distinguishable time unit by the system

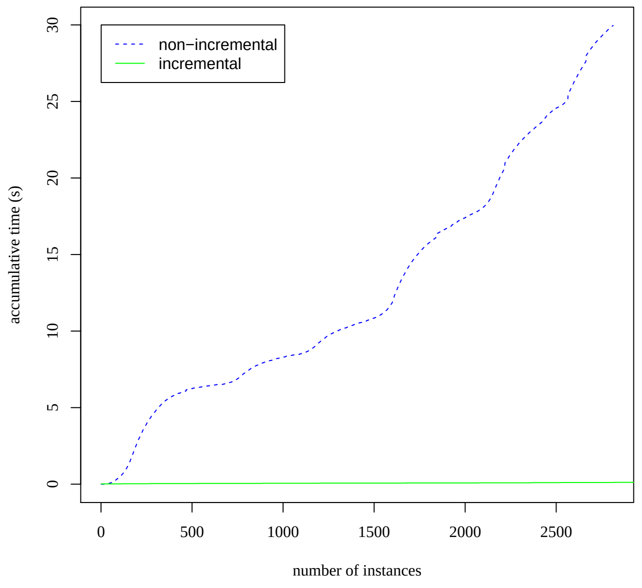

Figure 8.3.1: Incremental vs Nonincremental COM006-1

profile the accumulative time for the first n instances generated and use R to plot the generated sequence. Figure 8.3.1 shows the comparison of the accumulative time of approximately the first 3000 instances generated by the incremental versions of incOSHL and the nonincremental version of OSHL. We ran on problem COM006-1 and set the time limit to 30 seconds. The accumulative time of the nonincremental version is indicated by the dashed line. The accumulative time of the incremental version is indicated by the solid line. As we can see, the cost of generating new instances for the nonincremental version, because of repeated computation, increases much faster than the non incremental version. The nonincremental version only generated about 3000 instance within 30 seconds and could not prove the problem. In contrast, the incremental version generated about 1×10^6 instances within 30 seconds and found a proof for the problem. Figure 8.3.2 shows the behavior of the incremental version. As you can see, the accumulative cost is almost linear, meaning that the cost of generating new instances does not increase significantly with

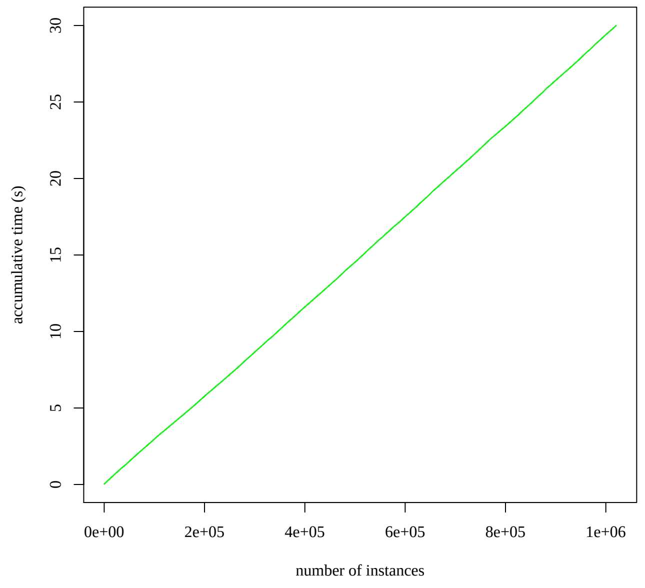

Figure 8.3.2: Incremental Behavior COM006-1

the number of instances already generated.

Figure 8.3.3 shows the results of tests on problem ANA002-1. The accumulative time of the nonincremental version is indicated by the dashed line. The accumulative time of the incremental version is indicated by the solid line. Not only does the cost of generating new instance for the nonincremental version increase much faster than the non incremental version, the speed of increase for the nonincremental version also increases as more instances are generated. The nonincremental version could not prove the problem. In contrast, the incremental version found a proof for the problem. Figure 8.3.4 shows the behavior of the incremental version. The accumulative cost is not linear any more, meaning that the cost of generating new instances increased with the number of instances already generated.

Figure 8.3.5 shows similar results on problem PLA004-1. The acculturative time of the nonincremental version is indicated by the dashed line. The accumulative time of the incremental version is indicated by the solid line. Cost of generating

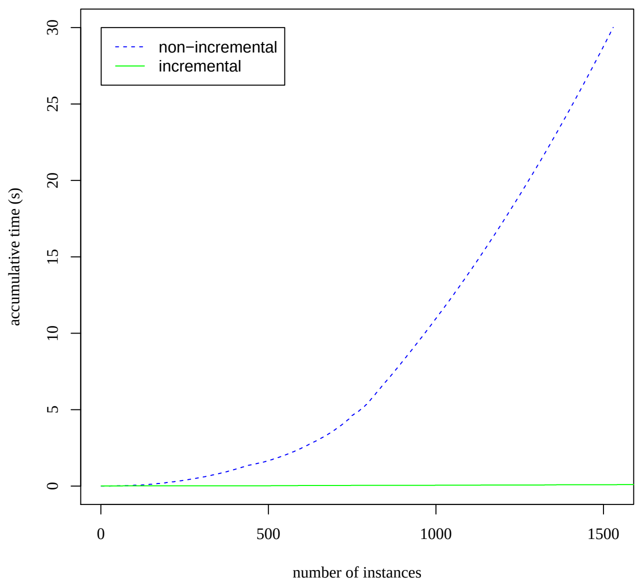

Figure 8.3.3: Incremental vs Nonincremental ANA002-1

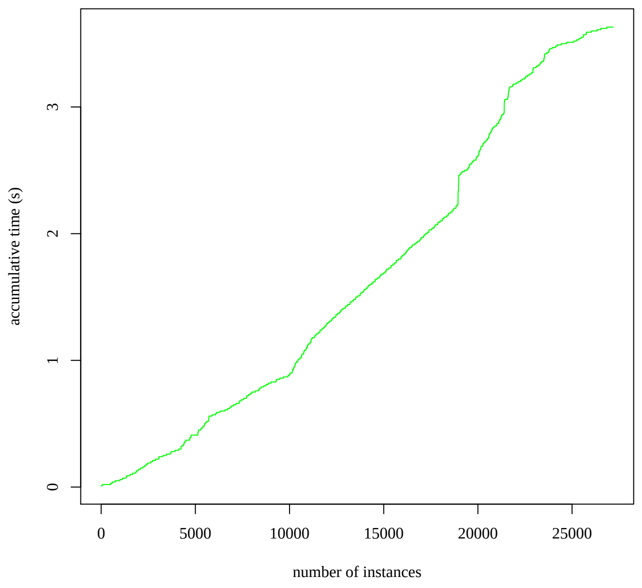

Figure 8.3.4: Incremental Behavior ANA002-1

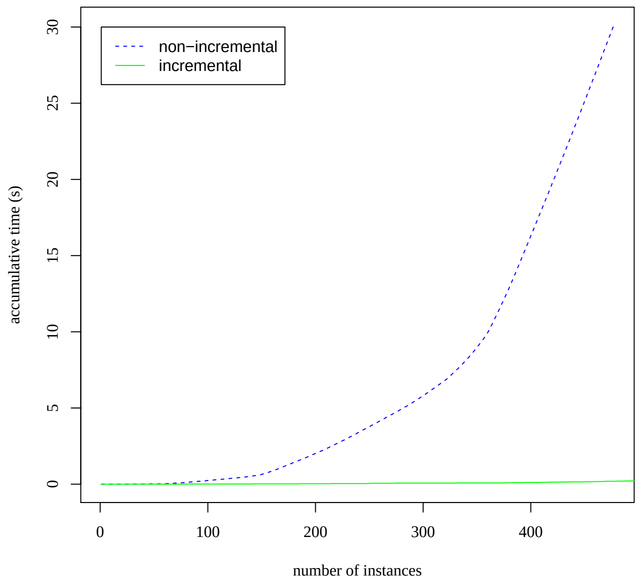

Figure 8.3.5: Incremental vs Nonincremental PLA004-1

new instance for the nonincremental version shows a super linear increase. Neither could prove the problem in 30 seconds. Figure 8.3.6 shows the behavior of the incremental version. The accumulative cost is not linear, either, meaning that the cost of generating new instances increased with the number of instances already generated.

On all of these three problems, we see at least 10 times speedup in the incremental version. The incremental version significantly improved the performance over the nonincremental version.

The incremental version of incOSHL is faster and uses more memory space than the non-incremental version of incOSHL. This is a space-speed trade-off. The original OSHL algorithm is space-efficient [28]. As a result, if we implement it in a non-incremental way, without introducing any cache, the implementation should also be space-efficient. However, as we have shown in this section, the performance of the non-incremental implementation is much worse than the performance of the

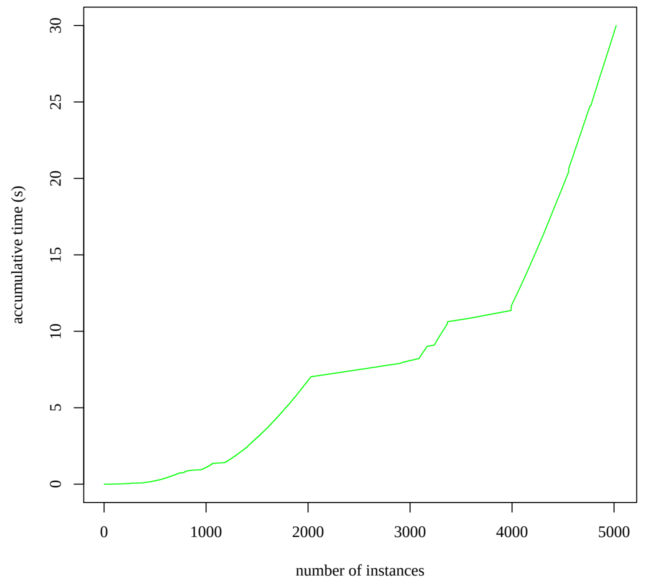

Figure 8.3.6: Incremental Behavior PLA004-1

incremental implementation on harder problems. The question then is what data we cache in the incremental implementation and whether they are necessary. A large chunk of data that we cache is program states, i.e., nodes with unvisited children and successful leaves in our decision tree. Nodes with unvisited children need to be cached because we may need to resume from there later; successful leaves need to be cached because an instance may be deleted from our set of instances and added back later. For every cached program state, we also need to cache all the activation records that lead to that program state. Most of these are needed to guarantee the optimality property of our implementation without mixing the semantics of incOSHL into the STACK EL. Even though these data are far less than a complete execution history of the program, it still constitutes a large cache. However, it should be noted that the size of the cache is always bounded by the size of the search space. Suppose that we limit our term size to a constant. Thus, our search space is also finite. If we restrict our decision tree to this search space, then we

	Untyped	Typed	Ratio
# Problems Solved	693	848	122.4%
Avg. # Inst. Generated	4.016×10^5	2.971×10^5	73.98%

Table 8.3: Untyped vs Typed: Proof Time and Number of Instances Generated

know that the size of the cache is bounded by the size of this restricted version of our decision tree. However, in our non-incremental implementation, this decision tree may be revisited as many times as the number of instances generated. Given this observation and that on the current hardware platform, it is easier to scale up main memory capacity than CPU speed, the space-time trade-off that we made in the incremental version of incOSHL seems reasonable.

8.4 Typed versus Untyped

The improvement of efficiency by incorporating type inference is measured in the following dimensions:

1. The number of problems that resulted in more than one type.
2. The number of problems solved in an untyped setting versus number of problems solved in a typed setting.
3. The reduction of number of instances.

For the first dimension, we ran the tests in the default setting. This way the prover will output the number of types resulting from type inference. Out of the 1528 problems tested, there are 795 problems that result in more than 1 types. That is, type inference affects 52.03% of the problems tested.

For the second dimension, we ran both the tests in the default setting and in a setting that is the same as the default one except that type inference is turned

TPTP Cat.	Total #	All neg. only	All pos. only	Both	Neither
ALG	1	0	0	1	0
ANA	16	4	0	6	6
CAT	1	0	0	1	0
COL	21	0	0	21	0
COM	8	0	1	7	0
FLD	161	55	0	8	98
GEO	3	0	0	2	1
GRA	1	0	0	1	0
GRP	21	14	0	7	0
HWV	11	8	0	3	0
KRS	9	0	0	9	0
LAT	11	2	0	9	0
LCL	285	8	0	26	251
MGT	22	7	0	15	0
MSC	18	1	0	14	3
NLP	7	0	0	7	0
NUM	19	4	0	8	7
PLA	43	6	20	3	14
PUZ	60	6	0	41	13
RNG	8	6	0	0	2
ROB	1	0	0	0	1
SET	45	8	0	33	4
SWV	143	11	1	62	69
SYN	608	67	0	353	188
TOP	5	1	0	3	1
Sum	1528	208	22	640	658

Table 8.4: All Negative versus All Positive

off. The result is shown in first row of Table 8.3. This table shows that with type inference the prover solves 22.4% more problems.

The second row of Table 8.3 shows that among the 690 problems that are solved under both the typed setting and the untyped setting, the prover generates 26.02% less instances under the typed setting than under the untyped setting.

8.5 Initial Interpretation

Previous work has shown that the choice of semantics can significantly affect the performance of the OSHL algorithm. For example, the original version of OSHL can solve 41% of SYN problems with all negative semantics and 71% of SYN problems with all positive semantics (on a significantly older TPTP version). [50] Our test results show that this is still true for incOSHL but to a lesser extent. We ran the tests twice, once using an initial interpretation that makes all negative literals true and once using an initial interpretation that makes all positive literals true. The results are shown in Table 8.4, categorized by TPTP problem categories. The first column shows the TPTP category; the second column shows the number of problems in our test set in each category; the third column shows the number of problems solved under the initial interpretation that makes all negative literals true but not under the initial interpretation that makes all positive literal true; the fourth column shows the converse; the fifth columns shows the number of problems solved under both initial interpretations; the sixth column shows the number of problems solved under neither interpretation. The summary row shows that there are 208 problem that are solved only under the initial interpretation that makes all negative literals true, 22 problems that are solved only under the initial interpretation that makes all positive literals true, 640 under both, and 658 under neither. The results show that the all negative initial interpretation outperforms the all positive initial interpretation. There is no essential difference between the two initial interpretations. The reason of the difference in performance is the particular way the test problems are formulated.

TPTP Cat.	Total #	w/o Rel. only	w/ Rel. only	Both	Neither
ALG	1	0	0	1	0
ANA	16	0	0	10	6
CAT	1	0	0	1	0
COL	21	0	0	21	0
COM	8	2	0	5	1
FLD	161	2	12	61	86
GEO	3	0	0	2	1
GRA	1	0	0	1	0
GRP	21	0	0	21	0
HWV	11	0	0	11	0
KRS	9	0	0	9	0
LAT	11	0	0	11	0
LCL	285	2	0	32	251
MGT	22	0	0	22	0
MSC	18	4	0	11	3
NLP	7	0	0	7	0
NUM	19	1	2	11	5
PLA	43	1	2	8	32
PUZ	60	2	3	45	10
RNG	8	1	1	5	1
ROB	1	0	0	0	1
SET	45	0	0	41	4
SWV	143	4	0	69	70
SYN	608	9	0	411	188
TOP	5	0	0	4	1
Sum	1528	28	20	820	660

Table 8.5: With Relevance versus Without Relevance

8.6 Relevance

We ran the tests twice, once using the default setting, and once using a setting that is the same as the default except relevance is turned on. The result is shown in Table 8.5, categorized by TPTP problem categories. The first column shows the TPTP category; the second column shows the number of problems in our test set in each category; the third column shows the number of problems solved without relevance but not with relevance; the fourth column shows the converse; the fifth

Problem	TPTP Rating
FLD059-1.p	0.1
LCL027-1.p	0.06
LCL187-1.p	0.11
LCL188-1.p	0.11
LCL360-1.p	0.06
NUM003-1.p	0.06
NUM004-1.p	0.06
NUM017-1.p	0.72
PLA006-1.p	0.06
PLA017-1.p	0.11
SET013-1.p	0.5
SET015-1.p	0.5
SYN639-1.p	0.44
SYN640-1.p	0.5
SYN646-1.p	0.44
SYN647-1.p	0.5

Table 8.6: Problems proved under increased memory limit

columns shows the number of problems solved with or without relevance; the sixth column shows the number of problems solved neither with nor without relevance. The summary row shows that there are 28 problems solved only without relevance, 20 problems solved only with relevance, 640 under both, and 658 under none. The results show that running without relevance outperforms running with relevance.

8.7 Increasing Memory Limit

Under the default setting, incOSHL proved 848 out of 1528 problem. Among the 680 problems that incOSHL failed to prove 437 reached the prover memory limit, 173 timed out, 67 exceed system resource limit, 2 reached term size limit, and one contains the “false literal” that is not supported by the current parser. For a majority of the problems that incOSHL failed to prove, it simply ran out of memory. We increased the LSF memory limit to 512G and incOSHL memory limit to 256G. Table 8.6 lists problems with TPTP rating > 0 which can be proved under this

	Total #	incOSHL	iProver	E	Darwin
ALG	1	1	1	1	1
ANA	16	12	14	15	8
CAT	1	1	1	1	1
COL	21	21	21	21	21
COM	8	8	8	5	6
FLD	161	77	115	127	100
GEO	3	2	2	2	2
GRA	1	1	1	1	1
GRP	21	21	21	21	21
HWV	11	11	11	11	11
KRS	9	9	9	9	9
LAT	11	11	11	11	11
LCL	285	34	200	264	260
MGT	22	22	22	22	22
MSC	18	15	13	12	14
NLP	7	7	7	7	7
NUM	19	14	19	19	18
PLA	43	29	34	35	16
PUZ	60	50	53	52	52
RNG	8	7	7	8	7
ROB	1	0	0	0	0
SET	45	41	45	45	44
SWV	143	74	137	75	97
SYN	608	420	607	601	563
TOP	5	4	5	5	0
Sum	1528	897	1364	1370	1292

Table 8.7: Theorem Provers Comparison (NOPP)

configuration. The majority of the rest of the problems timed out.

8.8 Comparison with Other Theorem Provers

We compare incOSHL with leading state-of-art theorem provers that are available to the public domain. Leading state-of-the-art theorem provers are provers that have won at least one CASC competition title and perform better than average state-of-the-art theorem provers. The purpose of this comparison is to show

how much difference there is between incOSHL and other leading state-of-the-art theorem provers.

Compared to leading state-of-the-art prover implementations, incOSHL is a relative new comer. It is inevitable that incOSHL lacks certain key features that are available on other theorem provers that have been under development for years. One feature that incOSHL lacks is a sophisticated set of strategies and strategy selection. Many theorem provers provide a large combination of different strategies that can be tweaked to work with different problem categories. A strategy selection mechanism allows the theorem prover to analyze the input problem and automatically choose a combination of strategies. A simplistic example would be to use an instance-based strategy for non-Horn problems and to use a resolution-based proof strategy for Horn problems. In contrast, incOSHL, as its predecessors in the OSHL prover family, uses a uniform strategy on all problems. Another characteristic of many leading theorem provers is that they are optimized toward the existing TPTP Library, and as a result it is difficult to separate performance due to the strategy from performance due to the fine tuning against the TPTP Library [43]. Optimization towards particular problem categories in the TPTP Library could be also extremely helpful in achieving good results in the CASC competition. For example, in CASC-23 (2011), out of 75 problem randomly selected for the EPT (effectively propositional theorem) division, 40 of them belong to SWV and 27 of them belong to SYN. [2] This is a result of these two categories having the majority of problems that qualify for the EPT division. In practice this could easily lead to bias towards a certain type of problems when doing prover optimization. Of course, this is not to say that a theorem prover optimized to perform better in particular divisions necessarily does not perform well in other divisions. In fact, years of CASC results have shown that the winner of one division usually performs reasonably well in some other divisions as well. In contrast, incOSHL has not been optimized towards the TPTP Library. This is consistent with our goal of developing an implementation of the OSHL algorithm using efficient data structures and subroutines and to provide a maintainable prover framework, so that we can study the performance of the basic OSHL algorithm and provide a platform for additional development and exploration.

In previous work [27, 50], Otter, a unification-based theorem prover with a single uniform strategy, was the choice of “standard of comparison” in the evaluation of various OSHL-based implementations and extensions. However, Otter development has been discontinued in 2004, hence it may not represent the state of the art. Because of a lack of “standard of comparison,” we instead simulate an optimal strategy selection by letting incOSHL run with the following four configurations and combining the results such that if a problem is proved under any configuration, it is counted as proved by incOSHL:

1. Initial interpretation that makes all negative literals true, without relevance (848)
2. Initial interpretation that makes all negative literals true, with relevance (840)
3. Initial interpretation that makes all positive literals true, without relevance (662)
4. Initial interpretation that makes all positive literals true, with relevance (731)

Although this somewhat compensates for a lack of strategy selection mechanism, it does not fully offset the lack of a sophisticated combination of strategies in incOSHL. After all, incOSHL still runs on a uniform strategy.

For our experiment, we also want to offset the effort of optimization towards larger TPTP categories, since incOSHL is not optimized for the TPTP Library. As readers may have already noticed, among the 25 TPTP categories in our test set, different TPTP categories have different numbers of problems. Ideally, we would like each category to have the same number of problems. One possibility would be to randomly choose in bigger categories a subset that matches the size of smaller categories. However, this would limit our problems selected to fewer problems and to the size of the minimum category. Simply discarding smaller categories would reduce the variety of kinds of problems that we test on. Therefore, we measure the performance on a category basis by calculating the following efficiency index (EI). Let N_i be the total number of problems in the ith category, S_i be the number of problems solved in that category, and N be the total number of categories. The efficiency index is computed as follows

$$\frac{\sum_{i=1}^{N} \frac{S_i}{N_i}}{N}$$

The theorem provers that we compare with include:

- iProver [20] is a theorem prover that combines instance-based inference using a SAT solver with resolution . It won the EPR [45] division at CASC-23 (2011), CASC-J5 (2010), CASC-22 (2009) and CASC-J4 (2008).
- E [40] is a saturation-based theorem prover for first-order logic with equality. It won the CNF [45] division at CASC-23 (2011) and received the "best overall" special award at CASC-J5 (2010).
- Darwin is an instance-based theorem prover based on the model evolution calculus [9], which lifts the propositional procedure DPLL to first-order logic. Darwin won the ERP division in CASC-21 (2007) and CASC-J6(2006). Darwin is no longer actively maintained since 2010.

	Max	incOSHL	iProver	E	Darwin
ALG	1.00	1.00	1.00	1.00	1.00
ANA	1.00	0.75	0.88	0.94	0.50
CAT	1.00	1.00	1.00	1.00	1.00
COL	1.00	1.00	1.00	1.00	1.00
COM	1.00	1.00	1.00	0.63	0.75
FLD	1.00	0.48	0.71	0.79	0.62
GEO	1.00	0.67	0.67	0.67	0.67
GRA	1.00	1.00	1.00	1.00	1.00
GRP	1.00	1.00	1.00	1.00	1.00
HWV	1.00	1.00	1.00	1.00	1.00
KRS	1.00	1.00	1.00	1.00	1.00
LAT	1.00	1.00	1.00	1.00	1.00
LCL	1.00	0.12	0.70	0.93	0.91
MGT	1.00	1.00	1.00	1.00	1.00
MSC	1.00	0.83	0.72	0.67	0.78
NLP	1.00	1.00	1.00	1.00	1.00
NUM	1.00	0.74	1.00	1.00	0.95
PLA	1.00	0.67	0.79	0.81	0.37
PUZ	1.00	0.83	0.88	0.87	0.87
RNG	1.00	0.88	0.88	1.00	0.88
ROB	1.00	0.00	0.00	0.00	0.00
SET	1.00	0.91	1.00	1.00	0.98
SWV	1.00	0.52	0.96	0.52	0.68
SYN	1.00	0.69	1.00	0.99	0.93
TOP	1.00	0.80	1.00	1.00	0.00
Avg.	1.00	0.80	0.89	0.87	0.79

Table 8.8: Theorem Provers Comparison (EI)

The result are shown in Table 8.7. The comparison shows that although incOSHL is lagging behind leading state-of-the-art theorem provers in the total number of problems solved, it is comparable in performance in ANA, COL, COM, GRP, HWV, KRS, LAT, MGT, MSC, NLP, NUM, PLA, PUZ, RNG, SET, and TOP. Among these categories, ANA, COM, MSC, PLA, PUZ, and RNG have an average TPTP rating greater than 0.1. The four main categories in which incOSHL is lagging behind are FLD, LCL, SWV, and SYN. The efficiency index is shown in Table 8.8. The average efficiency index of incOSHL among 25 TPTP categories is

	Total #	incOSHL	iProver	E	Darwin
ALG	1	1	1	1	1
ANA	9	6	8	8	2
CAT	1	1	1	1	1
COM	6	6	6	3	4
FLD	161	77	115	127	100
GRA	1	1	1	1	1
HWV	11	11	11	11	11
KRS	8	8	8	8	8
LAT	1	1	1	1	1
LCL	3	3	3	3	3
MGT	11	11	11	11	11
MSC	9	9	9	8	9
NLP	7	7	7	7	7
NUM	8	7	8	8	8
PLA	2	2	2	2	2
PUZ	36	34	36	36	35
SET	30	26	30	30	29
SWV	107	39	101	39	62
SYN	299	152	298	295	276
TOP	5	4	5	5	0
Sum	716	406	662	605	571

Table 8.9: Theorem Provers Comparison (NOPP, non-Horn Problems)

0.80, compared to 0.89 of iProver, 0.87 of E, and 0.79 of Darwin.

An important category of problems are the Horn problems. Horn problems are a subset of first-order logic problems that can be efficiently decided from the syntax of the problems. They are also a subset of UR-resolvable problems. Both SLD-Resolution [14] and UR-Resolution are complete on Horn problems. Among the 285 problems in LCL, only three are non-Horn problems.[3] These problems can be efficiently solved by employing a completely separate strategy based on resolution,

[3]Coincidentally, the categories in which incOSHL lags behind leading state-of-the-art theorem provers have the most number of problems. It is possible that the leading theorem provers has been heavily optimized towards these categories.

	MAX	incOSHL	iProver	E	Darwin
ALG	1.00	1.00	1.00	1.00	1.00
ANA	1.00	0.67	0.89	0.89	0.22
CAT	1.00	1.00	1.00	1.00	1.00
COM	1.00	1.00	1.00	0.50	0.67
FLD	1.00	0.48	0.71	0.79	0.62
GRA	1.00	1.00	1.00	1.00	1.00
HWV	1.00	1.00	1.00	1.00	1.00
KRS	1.00	1.00	1.00	1.00	1.00
LAT	1.00	1.00	1.00	1.00	1.00
LCL	1.00	1.00	1.00	1.00	1.00
MGT	1.00	1.00	1.00	1.00	1.00
MSC	1.00	1.00	1.00	0.89	1.00
NLP	1.00	1.00	1.00	1.00	1.00
NUM	1.00	0.88	1.00	1.00	1.00
PLA	1.00	1.00	1.00	1.00	1.00
PUZ	1.00	0.94	1.00	1.00	0.97
SET	1.00	0.87	1.00	1.00	0.97
SWV	1.00	0.36	0.94	0.36	0.58
SYN	1.00	0.51	1.00	0.99	0.92
TOP	1.00	0.80	1.00	1.00	0.00
Sum	1.00	0.88	0.98	0.92	0.85

Table 8.10: Theorem Provers Comparison (EI, non-Horn Problems)

which incOSHL lacks but can be easily incorporated. The comparisons on non-Horn problems are shown in Table 8.9. Again, we compute the efficiency index on all non-Horn problem, as shown in table 8.10. Compared with that of all problems, including Horn and non-Horn problems, incOSHL increased 0.08, iProver increased 0.09, E increased 0.05, and Darwin increased 0.06. The increase is probably the result of changing the problem set to its non-Horn subset. Nevertheless, the difference in the changes is interesting, E has increased the least. This may have shown that instance-based theorem provers perform better on non-Horn problems than saturation-based theorem provers.

Even without a sophisticated set of strategies, incOSHL is still able to prove some interesting problems, as shown in Table 8.11. We only included problems

TPTP Name	Rating
ANA002-1.p	0.8
ANA002-2.p	0.7
ANA002-3.p	0.8
ANA002-4.p	0.6
COM003-1.p	0.5
COM005-1.p	0.8
COM006-1.p	0.9
FLD011-3.p	0.6
MSC015-1.022.p	0.67
PLA004-1.p	0.56
PLA004-2.p	0.56
PLA005-1.p	0.56
PLA005-2.p	0.56
PLA009-1.p	0.56
PLA009-2.p	0.56
PLA011-1.p	0.56
PLA011-2.p	0.56
PLA013-1.p	0.56
PLA014-1.p	0.56
PLA014-2.p	0.56
PLA021-1.p	0.56
PLA031-1.006.p	0.67
RNG001-2.p	0.72
SET012-1.p	0.6
SYN802-1.p	0.6
SYN894-1.p	0.67

Table 8.11: Harder problems proved by incOSHL

with TPTP rating ≥ 0.5.

To conclude this section, we look at how well incOSHL performs on harder problems (TPTP rating greater than) compared to other theorem provers. The number of problems proved is shown in Table 8.12 and the efficiency index on harder problems is shown in Table 8.13. incOSHL has an efficiency index of 0.67, while iProver has 0.82, E has 0.79, and Darwin has 0.68. If incOSHL uses a simple strategy selection that offloads horn problems in LCL to a saturation-based strategy which results in an efficiency index of 0.7 (the lowest among the three compared

	Total #	incOSHL	iProver	E	Darwin
ANA	16	12	14	15	8
COL	21	21	21	21	21
COM	5	5	5	2	3
FLD	146	62	100	112	85
GEO	2	1	1	1	1
GRP	20	20	20	20	20
HWV	2	2	2	2	2
LAT	3	3	3	3	3
LCL	283	32	198	262	258
MGT	10	10	10	10	10
MSC	9	6	4	4	5
NUM	11	7	11	11	10
PLA	39	25	30	31	12
PUZ	20	11	13	13	12
RNG	8	7	7	8	7
ROB	1	0	0	0	0
SET	18	15	18	18	17
SWV	96	40	90	40	51
SYN	186	93	185	181	141
TOP	2	1	2	2	0
Sum	898	373	734	756	666

Table 8.12: Theorem Provers Comparison (NOPP, TPTP rating > 0)

provers), then the average efficiency index would be 0.7.

The comparison on harder non-Horn problems is shown in Table 8.14 and Table 8.15. On harder non-Horn problems, incOSHL has an efficiency index of 0.76, while iProver has 0.96, E has 0.83, and Darwin has 0.69.

	Max	incOSHL	iProver	E	Darwin
ANA	1.00	0.75	0.88	0.94	0.50
COL	1.00	1.00	1.00	1.00	1.00
COM	1.00	1.00	1.00	0.40	0.60
FLD	1.00	0.42	0.68	0.77	0.58
GEO	1.00	0.50	0.50	0.50	0.50
GRP	1.00	1.00	1.00	1.00	1.00
HWV	1.00	1.00	1.00	1.00	1.00
LAT	1.00	1.00	1.00	1.00	1.00
LCL	1.00	0.11	0.70	0.93	0.91
MGT	1.00	1.00	1.00	1.00	1.00
MSC	1.00	0.67	0.44	0.44	0.56
NUM	1.00	0.64	1.00	1.00	0.91
PLA	1.00	0.64	0.77	0.79	0.31
PUZ	1.00	0.55	0.65	0.65	0.60
RNG	1.00	0.88	0.88	1.00	0.88
ROB	1.00	0.00	0.00	0.00	0.00
SET	1.00	0.83	1.00	1.00	0.94
SWV	1.00	0.42	0.94	0.42	0.53
SYN	1.00	0.50	0.99	0.97	0.76
TOP	1.00	0.50	1.00	1.00	0.00
Average	1.00	0.67	0.82	0.79	0.68

Table 8.13: Theorem Provers Comparison (EI, TPTP rating > 0)

	Total #	incOSHL	iProver	E	Darwin
ANA	9	6	8	8	2
COM	3	3	3	0	1
FLD	146	62	100	112	85
HWV	2	2	2	2	2
LCL	1	1	1	1	1
MSC	3	3	3	3	3
NUM	1	1	1	1	1
PLA	1	1	1	1	1
PUZ	4	3	4	4	3
SET	12	9	12	12	11
SWV	71	16	65	15	27
SYN	117	63	116	115	94
TOP	2	1	2	2	0
Sum	372	171	318	276	231

Table 8.14: Theorem Provers Comparison (NOPP, TPTP rating > 0, non-Horn)

	Max	incOSHL	iProver	E	Darwin
ANA	1.00	0.67	0.89	0.89	0.22
COM	1.00	1.00	1.00	0.00	0.33
FLD	1.00	0.42	0.68	0.77	0.58
HWV	1.00	1.00	1.00	1.00	1.00
LCL	1.00	1.00	1.00	1.00	1.00
MSC	1.00	1.00	1.00	1.00	1.00
NUM	1.00	1.00	1.00	1.00	1.00
PLA	1.00	1.00	1.00	1.00	1.00
PUZ	1.00	0.75	1.00	1.00	0.75
SET	1.00	0.75	1.00	1.00	0.92
SWV	1.00	0.23	0.92	0.21	0.38
SYN	1.00	0.54	0.99	0.98	0.80
TOP	1.00	0.50	1.00	1.00	0.00
Average	1.00	0.76	0.96	0.83	0.69

Table 8.15: Theorem Provers Comparison (EI, TPTP rating > 0, non-Horn)

Chapter 9

Conclusion

On the theoretical level, we introduced genOSHL, an abstract, generalized version of OSHL which captures the essential features of OSHL. We proved the soundness and completeness of genOSHL. Then, we introduced incOSHL, a specialized version of genOSHL, which differs from the original OSHL algorithm. We also introduced a type inference algorithm which allows genOSHL to possibly reduce its search space while still preserving the soundness and completeness.

On the practical level, we designed and implemented a low-level framework that combines coroutines with dependency of program states on global mutable data. This framework is exposed to programmers in a well-defined embedded programming language called the STACK EL. The STACK EL allows relatively easy modification to the source code while still preserving key properties that ensure high performance. We also introduced our implemented incOSHL with type inference in the STACK EL in a relatively detailed form of pseudo-code. We described a simpler, non-incremental implementation and how we applied various programming constructs provided by the STACK EL to create an optimal, incremental implementation. The incremental version of incOSHL has much improved performance over previous generation OSHL. By incorporating the STACK EL, and including efficient implementations of many key data structures in C++, incOSHL has laid a solid foundation for future OSHL work.

We also studied the performance of our incremental theorem prover on a set of test problems chosen from a wide variety of categories. We showed that OSHL can be implemented efficiently so that it has relatively high inference rate. We showed that our prover is capable of proving about half of the problems in our test sets. We showed that type inference helps both reduce the search space and improve the NOPP. We compared our theorem prover with leading state-of-the-art theorem provers. We showed that the efficiency index of our theorem prover is comparable to other state-of-the-art theorem provers, despite a lack of sophisticated combinations of strategies.

Possible future research includes:

1. Special handling of unit clauses: Unit input clauses and any unit clauses derived from input clauses using resolution have the property that their complements can never be true in a model of the input clauses. This suggests that we can handle unit clauses in a way that differs from how we handle non-unit clauses. Once we find a unit clause, we can immediately modify the model, and never revoke that modification during the proof search. On the one hand, this may allow more efficient data structures, since the delete operation is not needed for unit clauses. On the other hand, this may allow more efficient caching, since backtracking is not needed for unit clauses.

2. Developing more strategies and a strategy selection mechanism: Future research may include incorporating special strategies for special classes of problems (such as the Horn problems) and a mechanism for detecting problems that are suitable for the special strategies and dispatching those problems to the special strategies. For example, the performance of our theorem prover can be significantly improved if we combine our relatively uniform strategy

with a resolution-based strategy, such as unit resulting resolution or hyper-resolution, for Horn problems.

3. Equality: Many TPTP problems contain a special predicate for equality. Equality could be handled axiomatically, but it is not efficient. Usually, proof strategies are extended with direct support for equality for better performance. incOSHL currently does not have support for equality. Adding support for equality to the proof strategy may improve the performance of incOSHL on problems with equality.

4. More refined types: Our type inference algorithm may infer for every variable a subset of all terms which are used to instantiate that variable. One interesting question would be: Can we further reduce that subset, thereby reducing the search space of our proof strategy?

5. Understanding the variation of inference rate on different problems: As we have shown in our performance study, incOSHL has a high average inference rate, but on some of the problems, the inference rate is much lower than the average inference rate. It would be interesting to looking into those problems and find out why the inference rate is low for those problems.

6. More sophisticated semantics: One of the features of the original OSHL algorithm is that it allows manually adding sophisticated semantics. Although the incOSHL proof strategy allows the same level of support for sophisticated semantics, it is not implemented in the current version of the theorem prover. Future research may include finding an efficient way to implement support for sophisticated semantics and automatic semantic generation.

7. Reducing cache space: One of the problems of our implementation of incOSHL is that the incremental version consumes a lot of memory space. Some of

the space used by the cache can be reduced if the lower-level system, i.e., the generic embedded language STACK EL is aware of the semantics of the higher-level system, i.e., the incOSHL data structures and subroutines, and makes use of the semantics to discard unused cache. A naive injection of the incOSHL semantics into the STACK EL would blur our stratification and would reduce modularity of the code. Future research may include finding a clean, modular way to integrate application-specific semantics into the STACK EL.

8. Trying the prover on more problems: Future research may include trying the prover on problems not in the TPTP library and fine tuning the prover for the CASC competition.

References

[1] Getting started on killdevil. http://help.unc.edu/CCM3_031537.

[2] selected problems. http://www.cs.miami.edu/ tptp/CASC/23/ SelectedProblems.html.

[3] Tptp document file: Overallsynopsis. http://www.cs.miami.edu/ tptp/cgi-bin/SeeTPTP?Category=Documents&File=OverallSynopsis.

[4] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(01):47–74, 1996.

[5] Franz Baader, Andrei Voronkov, Franz Baader, and Andrei Voronkov. Abstract DPLL and Abstract DPLL Modulo Theories Logic for Programming, Artificial Intelligence, and Reasoning. volume 3452 of *Lecture Notes in Computer Science*, chapter 3, pages 36–50. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[6] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Superposition with simplification as a desision procedure for the monadic class with equality. In *Proceedings of the Third Kurt Godel Colloquium on Computational Logic and Proof Theory*, KGC '93, pages 83–96, London, UK, UK, 1993. Springer-Verlag.

[7] Peter Baumgartner. Fdpll – a first-order davis-putnam-logeman-loveland procedure. In *CADE-17 – The 17th International Conference on Automated Deduction, volume 1831 of Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.

[8] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Darwin: A Theorem Prover for the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR (aka S4))*, Electronic Notes in Theoretical Computer Science, 2004.

[9] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. pages 350–364. 2003.

[10] Daniel G. Bobrow and Ben Wegbreit. A model and stack implementation of multiple environments. *Commun. ACM*, 16(10):591–603, October 1973.

[11] Koen Claessen. Equinox, A New Theorem Prover for Full First-Order Logic with Equality. 2005.

[12] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[13] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[14] Jean H. Gallier and Stan Raatz. Extending SLD resolution to equational horn clauses using e-unification. *The Journal of Logic Programming*, 6(1-2):3–43, January 1989.

[15] Harald Ganzinger, Konstantin Korovin, and Phokion Kolaitis. New directions in instantiation-based theorem proving. 2003.

[16] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, 37:5–20, 1996.

[17] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. 2008.

[18] Aleksey Gurtovoy and David Abrahams. The boost c++ metaprogramming library, 2002.

[19] J. P. Kearns, Carol J. Meier, and Mary Lou Soffa. The performance evaluation of control implementations. *IEEE Trans. Softw. Eng.*, 8(2):89–96, March 1982.

[20] Konstantin Korovin. iprover — an instantiation-based theorem prover for first-order logic (system description). In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 292–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Konstantin Korovin. Instantiation-based automated reasoning: From theory to practice. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 163–166, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3-4):227–260, December 1971.

[23] Reinhold Letz and Gernot Stenz. The Disconnection Tableau Calculus. *Journal of Automated Reasoning*, 38(1-3):79–126, December 2006.

[24] D. W. Loveland. A linear format for resolution Symposium on Automatic Demonstration. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, chapter 10, pages 147–162. Springer Berlin / Heidelberg, 1970.

[25] David Luckham. Refinement theorems in resolution theory Symposium on Automatic Demonstration. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, chapter 11, pages 163–190. Springer Berlin / Heidelberg, 1970.

[26] William McCune. Otter 2.0. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 663–664, London, UK, UK, 1990. Springer-Verlag.

[27] Swaha Miller. OSHL-U: A First-Order Theorem Prover Using Propositional Techniques and Semantics, 2005.

[28] Swaha Miller and David A. Plaisted. The Space Efficiency of OSHL. In *TABLEAUX*, 2005.

[29] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving, 2001.

[30] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[31] David A. Plaisted. The search efficiency of theorem proving strategies. In *Proceedings of the 12th International Conference on Automated Deduction*, CADE-12, pages 57–71, London, UK, UK, 1994. Springer-Verlag.

[32] David A. Plaisted and Gregory Kucherov. The complexity of some complementation problems. *Information Processing Letters*, 71(3-4):159–165, August 1999.

[33] David A. Plaisted and Yunshan Zhu. Ordered semantic hyper linking. In *Journal of Automated Reasoning*, 2000.

[34] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, August 2002.

[35] George Robinson and Lawrence Wos. PARAMODULATION AND THEOREM PROVING IN FIRST ORDER THEORIES WITH EQUALITY. 1968.

[36] J. Robinson. *Handbook of Automated Reasoning: Volume 1.* MIT Press, Cambridge, MA, USA, 2001.

[37] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[38] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[39] J. A. Robinson. Automatic deduction with hyper-resolution. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pages 416–423. Springer, Berlin, Heidelberg, 1983.

[40] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[41] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Handbook of automated reasoning. chapter Term indexing, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[42] James R. Slagle. Automatic theorem proving with renamable and semantic resolution. *J. ACM*, 14(4):687–697, October 1967.

[43] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[44] Geoff Sutcliffe. The tptp problem library tptp v5.0.0. http://www.cs.miami.edu / tptp/TPTP/TR/TPTPTR.shtml.

[45] Geoff Sutcliffe. CASC-J3 The 3rd IJCAR ATP System Competition Automated Reasoning. volume 4130 of *Lecture Notes in Computer Science*, chapter 46, pages 572–573. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[46] Eric W. Weisstein. Lexicographic order. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LexicographicOrder.html.

[47] Lawrence Wos, Daniel Carson, and George Robinson. The unit preference strategy in theorem proving. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, AFIPS '64 (Fall, part I), pages 615–621, New York, NY, USA, 1964. ACM.

[48] Lawrence Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, October 1965.

[49] Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *J. ACM*, 14(4):698–709, October 1967.

[50] Yunshan Zhu. Efficient First-Order Semantic Deduction Techniques, 1998.