Masthead Logo

University of Iowa
Iowa Research Online

Theses and Dissertations

Fall 2013

Finite model finding in satisfiability modulo theories

Andrew Joseph Reynolds
University of Iowa

Copyright 2013 Andrew J. Reynolds

This dissertation is available at Iowa Research Online: https://ir.uiowa.edu/etd/5047

Recommended Citation
Reynolds, Andrew Joseph. "Finite model finding in satisfiability modulo theories." PhD (Doctor of Philosophy) thesis, University of Iowa, 2013.
https://doi.org/10.17077/etd.mvb1eu00

Follow this and additional works at: https://ir.uiowa.edu/etd

Part of the Computer Sciences Commons

FINITE MODEL FINDING IN SATISFIABILITY MODULO THEORIES

by

Andrew Joseph Reynolds

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2013

Thesis Supervisor: Professor Cesare Tinelli

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Andrew Joseph Reynolds

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2013 graduation.

Thesis Committee: ______________________________
Cesare Tinelli, Thesis Supervisor

____
Aaron Stump

____
Hantao Zhang

____
Sriram Pemmaraju

____
Clark Barrett

For my wife and family.

ACKNOWLEDGEMENTS

Many thanks to people in my life that made this possible. I would like to thank my parents, Albert and Paige, and my sister Julie for all of their love and support. A special thanks to my wife Marina for her patience and love for me. I would like to thank my advisor, Cesare Tinelli, whose attention to detail and high standards for technical writing have been invaluable to me. Many thanks to Aaron Stump, whose course at Washington University helped inspire me to pursue a career in research. I would like to thank Amit Goel and Sava Krstić from the Intel Corporation for their insightful collaboration on this work. I would also like to thank Leonardo de Moura for his time spent as an advisor to me at Microsoft Research, and whose technical discussions inspired some of the key ideas in this thesis. Finally, I would like to thank the development team of CVC4, especially Clark Barrett and Morgan Deters, for their ingenuity and bug fixes over the past several years.

TABLE OF CONTENTS

LIST OF FIGURES vii

CHAPTER

1 INTRODUCTION 1

1.1 Satisfiability Modulo Theories 1
1.2 Contributions 4

2 SATISFIABILITY AND SATISFIABILITY MODULO THEORIES . 6

2.1 The Satisfiability Problem 6
2.2 Satisfiability Modulo Theories 9
2.3 Applications 12

3 HANDLING QUANTIFIED FORMULAS 14

3.1 Quantified Formulas in SMT 14
3.1.1 Pattern-Based Instantiation 17
3.1.2 Complete Instantiation 21
3.1.3 Model-Based Quantifier Instantiation 22
3.1.4 Quantifier Elimination 24
3.2 First-Order Theorem Proving 25
3.2.1 Inst-Gen 28
3.2.2 Finite Model Finding 30
3.2.2.1 MACE-Style Model Finding 30
3.2.2.2 SEM Model Finding 32

4 FORMAL PRELIMINARIES 34

4.1 Preliminaries 34
4.2 DPLL(T) Procedure 37
4.2.1 A Typical Strategy for DPLL(T) 43

5 FINITE MODEL FINDING IN SMT 45

5.1 A Model-Based Approach for Quantifiers in SMT 46
5.2 EUF with Finite Cardinality Constraints (EFCC) 48
5.2.1 Decision Procedure 49
5.2.2 Integration into DPLL($T_1, \ldots, T_m$) 51

5.2.3 Efficient Solver . . . 53
5.2.3.1 Weak Effort Check . . . 53
5.2.3.2 Strong Effort Check . . . 57
5.2.4 Establishing Finite Cardinalities . . . 59
5.2.4.1 Extension to Multiple Sorts . . . 60
5.3 Constructing Candidate Models . . . 64
5.3.1 Choosing Domain Elements . . . 65
5.3.2 Representing Function Definitions . . . 67
5.3.3 Constructing Function Definitions . . . 70
5.3.4 Simplifying Function Definitions . . . 74
5.4 Model-Based Quantifier Instantiation . . . 75
5.4.1 Algorithm for Generalizing Evaluations . . . 76
5.4.1.1 Generalizing Evaluations . . . 77
5.4.1.2 Choosing Instantiations . . . 80
5.4.2 Algorithm for Computing Interpretations for Terms . . . 83
5.4.2.1 Operations on Definitions . . . 83
5.4.2.2 Computing Interpretations for Terms . . . 89
5.4.2.3 Choosing Instantiations . . . 92
5.4.3 Integration into DPLL($T_1, \ldots, T_m$) . . . 95
5.5 Properties . . . 96
5.5.1 Finite Model Completeness . . . 96
5.5.2 Refutational Completeness . . . 97
5.6 Enhancements . . . 101
5.6.1 Heuristic Instantiation . . . 101
5.6.2 Sort Inference . . . 102
5.6.3 Relevancy . . . 108
5.7 Results . . . 108
5.7.1 EFCC Solver Evaluation . . . 109
5.7.2 Finite Model Finder Evaluation . . . 111
5.7.2.1 Intel benchmarks . . . 113
5.7.2.2 TPTP benchmarks . . . 116
5.7.2.3 Isabelle benchmarks . . . 120

6 EXTENSIONS TO OTHER DOMAINS . . . 124

6.1 Bounded Integer Quantification . . . 124
6.1.1 Inferring Bounds . . . 125
6.1.2 Establishing Finite Bounds . . . 126
6.1.3 Constructing Candidate Models . . . 126
6.1.3.1 Representing Function Definitions . . . 127
6.1.3.2 Constructing Function Definitions . . . 127
6.1.4 Quantifier Instantiation . . . 129
6.1.5 Properties . . . 131
6.1.6 Results . . . 133

6.2 Strings 134

7 CONCLUSION 136

APPENDIX

A PREPROCESSING 139

A.1 Negation Normal Form 139
A.2 Miniscoping 140
A.3 Destructive Equality Resolution 140
A.4 Eliminating Nested Quantifiers 140
A.5 Convert to a Set of Clauses 141

B EXTENSIONS TO MODEL-BASED QUANTIFIER INSTANTIATION 142

REFERENCES 145

LIST OF FIGURES

Figure

3.1 Rules for Resolution-Based Theorem Proving 26

3.2 Rule for Paramodulation 27

3.3 Rule for Inst-Gen 29

4.1 DPLL($T_1, \ldots, T_m$) rules 39

4.2 A typical strategy check for applying DPLL($T_1, \ldots, T_m$) rules 43

5.1 The fix_region procedure 56

5.2 The eval procedure 77

5.3 The choose_instances procedure 81

5.4 Method for computing composition of entries 86

5.5 DPLL($T_1, \ldots, T_m$) rule for quantifier instantiation. 95

5.6 Results for randomly generated benchmarks 110

5.7 Results for satisfiable and unsatisfiable Intel (DVF) benchmarks 114

5.8 Results for TPTP benchmarks. 117

5.9 Satisfiable TPTP problems with and without model-based instantiation. 119

5.10 Results for satisfiable and unsatisfiable Isabelle benchmarks 122

6.1 The infer_bounds procedure 125

6.2 The bound_int_qi procedure 129

6.3 Results for Intel benchmarks containing bounded integer quantification . 133

B.1 Extended method for computing composition of entries 143

CHAPTER 1

INTRODUCTION

In recent years, automated reasoning has had a growing impact on software engineering practices. As often seen, errors in hardware and software systems can lead to unforeseen consequences, costing governments and corporations millions of dollars, and in some cases leading to loss of life. Consequently, there has been a high demand for software that is verified using formal methods. These methods are of utmost importance to software whose correctness is highly critical, including software for managing bank records, flight control software, and software used in medical devices.

The field of formal methods has made large strides towards formally verifying software on a large scale, thanks in part to theoretical advancements of automated reasoning. In particular, many successful verification and synthesis applications in recent years have relied heavily upon the use of Satisfiability Modulo Theories (SMT) solvers for answering logical queries required for solving complex problems.

1.1 Satisfiability Modulo Theories

Many problems in first-order logic can be efficiently handled by modern Satisfiability Modulo Theories (SMT) solvers. In these problems, the interpretation of certain functions is constrained according to some background theory, including real and integer arithmetic, bit-vectors, arrays, uninterpreted functions with equality (EUF) and combinations thereof. While many of these problems have theoretical complexity that is worst-case exponential, SMT solvers have shown surprising efficiency for an

overwhelming majority of problems that occur in practice. This efficiency is obtained by exploiting the structure of such problems, which in many cases can effectively reduce the search space for solutions dramatically.

The use of an underlying SMT solver in user applications is attractive for a variety of reasons. Due to the wide range of problems that arise in formal methods applications, the expressive power of SMT is often useful when encoding logical queries for a particular task. Since many SMT solvers have built-in support for a number of theories, this often allows the user a more natural encoding as compared to, say, purely propositional encodings. Some applications, such as those relying on arithmetic, would be otherwise infeasible if not for an encoding that makes use of background theories.

The performance of SMT solvers has improved significantly in recent years, due in part to developments in Boolean Satisfiability (SAT) technology, as well as the development and implementation of efficient decision procedures for quantifier-free constraints in certain theories. For the former, most SMT solvers integrate an off-the-shelf SAT solver that incorporates many modern optimizations [47] [62], including efficient techniques for unit propagation, non-chronological backtracking, conflict-driven clause learning, decision heuristics, among others. Since most SMT solvers require little modification of their underlying SAT solvers, these techniques often translate into improved performance when considering problems containing both propositional structure and theory content.

Approaches for theory decision procedures have made significant advances in

the past decade, both theoretically and in terms of implementation. These include fast congruence closure for equality and uninterpreted functions [49], fast simplex approaches for linear arithmetic [24], and efficient approaches for the theory of extensional arrays [11, 30]. These decision procedures use methods for eagerly recognizing when a set of constraints is unsatisfiable, as well as when certain constraints may be propagated. Experimental evidence suggests that many of these theories are reaching a mature state, as performance in recent years has stabilized for several commonly used theories.

Due to the large number of theories supported by SMT solvers, a generalized method for determining satisfiability in problems containing a combination of theories is of practical interest. For quantifier-free formulas, most SMT solvers use the Nelson-Oppen combination procedure [48] in which decision procedures for (stably infinite) theories can be combined modularly, giving a decision procedure for the combined theory. As a result, SMT implementers need only focus on constructing solvers for a problem purified to contain only constraints in a single theory, with the guarantee that their solver can be incorporated into an approach for combined theories in a generalized and even efficient manner [33]. This has been highly important in extending the scope of SMT solvers, since most applications require the use of multiple theories.

Although many classes of SMT problems reside in decidable fragments of first-order logic, recent work has focused on undecidable classes of problems, including problems in certain fragments containing universal quantification [28] and non-linear integer arithmetic [34]. Devising a general approach for these problems has been an

ongoing challenge in the SMT community.

1.2 Contributions

In this thesis, we provide new methods for handling SMT problems containing universal first-order quantification. We will examine an approach known as *finite model finding*, which has been used successfully by the automated proving theorem community as a method for finding models of quantified first-order formulas. We provide a method for finite model finding in SMT that is finite model complete for a fragment of first-order logic that occurs commonly in practice. A secondary goal of this thesis is to improve upon state-of-the-art approaches for answering unsatisfiable. Most current approaches in SMT for establishing unsatisfiability in the presence of quantified formulas rely heavily on incomplete heuristics for quantifier instantiation. In contrast, the approach developed in this thesis is refutationally complete under certain restrictions. Additionally, we provide experimental evidence showing that our approach is practically feasible within various applications, including hardware and software verification, and automated theorem proving.

Overview In Chapter 2, we introduce the satisfiability problem and commonly used procedures for solving this problem. We also introduce its extension to Satisfiability Modulo Theories. In Chapter 3, we review various approaches from the SMT and automated theorem proving communities for handling first-order quantified formulas. Chapter 4 gives a formal introduction to notions and procedures used in the remainder of the thesis. In Chapter 5, we describe in detail a new approach for handling quantified formulas, finite model finding in SMT, which can be integrated into the

architecture commonly used by modern SMT solvers. We show several important properties of this approach, and provide experimental evidence that it is highly competitive with respect to both state of the art SMT solvers and model finders from the automated theorem proving community. In Chapter 6, we discuss how approaches similar to finite model finding in SMT can be extended to handle other domains of interest, including quantification over integers where finite bounds can be inferred. We provide preliminary evidence to show that our approach is feasible for this domain as well.

CHAPTER 2

SATISFIABILITY AND SATISFIABILITY MODULO THEORIES

2.1 The Satisfiability Problem

In Boolean logic, logical formulas are composed of propositional atoms from a fixed finite set P, the symbols **true** and **false**, and the standard logical connectives such as $\vee, \wedge, \Rightarrow$. Given the standard interpretation for these symbols, the Boolean satisfiability problem for a formula φ asks if there exists an assignment to the propositional atoms such that the φ evaluates to true.

For each propositional atom p in P, we refer to p or its negation $\neg p$ as a *literal*. We will commonly write $\bar{l}$ to denote the complement of literal l, e.g. p and $\neg p$ are complements of each other. We refer to a conjunction of literals $(l_1 \vee \ldots l_n)$ as a *clause*. A conjunction of clauses $C_1 \wedge \ldots C_n$ is a formula in conjunctive normal form (CNF). It is possible to convert any formula φ into an equisatisfiable formula in conjunctive normal form.

The DPLL procedure[1] is used by a majority modern SAT solvers [26] for determining the satisfiability of Boolean formulas in conjunctive normal form. Although SAT is a well known NP-complete problem, the optimizations used by modern SAT solvers have made it possible to answer such problems with surprising efficiency. In a common declarative formalism [50], the DPLL procedure is described as operating on states of the form $M \parallel F$, where M is a (initially empty) sequence of literals

[1] DPLL is named for its authors, Davis, Putnam, Logemann, and Loveland.

and F is a set of clauses, initially the input clauses. For each state $M \parallel F$ that is reachable by this procedure, we have that each atom occurs in at most one literal in M. Thus, the sequence M can be thought of as a partial assignment from atoms to truth values, and we will sometimes refer to literals l as being *assigned* in M if l or its complement occurs in M. We say that M is *complete* if all atoms in F are assigned in M. We say that a clause C is *satisfied* by M if at least one of its literals occurs in M. Dually, we say that a clause C is *falsified* if the complement of all of its literals occur in M. The DPLL procedure will search for a sequence M where all clauses in F are satisfied, in which case the problem is satisfiable. Otherwise, if the procedure determines that every complete sequence M falsifies at least one clause in F, the problem is unsatisfiable.

When searching for an M that satisfies F, the procedure adds literals to M in one of two ways. Firstly, if there exists a clause $C \vee l$ in F such that all literals in C are assigned to false in M, and l is unassigned in M, then we must add l to M. In this case, we say l is *asserted by propagation*. Secondly, the procedure can choose to add an arbitrary unassigned literal l to M, in which case we say l is *asserted as a decision*, and commonly refer to l as a *decision literal*. A typical strategy for DPLL asserts literals by propagation exhaustively before asserting any literal as a decision.

The procedure also monitors when a clause C in F becomes falsified by the current assignment M. In this case we call C a *conflicting clause*, and the procedure must remove some the literals from M. In the most basic implementation, it does so by finding the most recent decision literal l, removing l and all subsequent literals in

M, and asserting $\bar{l}$ as a propagation. If M contains no decision literals and there is a conflicting clause, then the problem is unsatisfiable.

Modern SAT solvers incorporate many optimizations beyond the basic DPLL procedure as described here. Most significantly, most solvers use non-chronological backtracking, where multiple decision literals can be backtracked at once when a conflicting clause is found. Conflict analysis techniques can be performed by maintaining an *implication graph*, which indicates which clauses were the source of propagations. This data structure can be used for determining how far the solver may backtrack when a conflict occurs. In this process, the solver will have determined some set of asserted literals that led us to encounter the conflict. The disjunction of the complement of these literals is referred to as a *conflict clause*. Conflict clauses can be learned, that is, added to the original clause set F, because they are implied by F. Their effect is to prune the search space of the problem, since each of these clauses imposes additional constraints on the truth assignment we are searching for. This process is known as conflict-driven clause learning (CDCL).

Efficient techniques can be used for detecting when propagation can be applied. The 2-watched literal approach can be used to recognize when an unsatisfied clause contains only one unassigned literal and therefore must be propagated [47, 60]. When choosing decision literals, heuristics can be used to judge which literal to choose next. These heuristics can be based on how often a literal appears in conflicts [47]. SAT solvers also have heuristics for judging the usefulness of learned clauses, for instance, by keeping track of how often they participate in propagations and conflicts. Since

the performance of the solver can be highly dependent upon the size and number of clauses we are considering, it is necessary that the solver manage the clauses occurring in F. Clauses can be unlearned if they are determined to be unhelpful, or all at once during a search restart if the number of learned clauses becomes too large.

2.2 Satisfiability Modulo Theories

In this section, we introduce techniques for determining the satisfiability of quantifier-free formulas with background theories, which build upon those described in the previous section. Whereas in the Boolean Satisfiability problem, our input formula consisted of propositional atoms, in the following, our input formula will consist of atoms taken from a signature Σ. A signature Σ consists of a set of function and predicate symbols, and a set of variables. For example, the formula $x+y \geq 0$ is an atom for the theory of arithmetic, where $\geq$ is a binary predicate. The satisfiability question when extended to theories is restricted by the interpretation of symbols in the signature of the theory. We consider a *theory* with signature Σ to be a set of deductively closed Σ-formulas, that is, formulas built using the symbols in Σ. We refer to function symbols occurring in T as *interpreted*, and all other function symbols as uninterpreted. In other words, conceptually a theory T contains the (possibly infinite) axiomatization of all interpreted symbols of that theory. Informally, a formula φ is satisfiable modulo a background theory T, or T-satisfiable, if there exists a model that satisfies both φ and the theory T.

Approaches for satisfiability modulo theories can be broken up into two categories, eager approaches and lazy approaches. Eager approaches convert a problem

containing theories into a equisatisfiable problem at the propositional level. For example, Ackermann's reduction can be used to eliminate uninterpreted functions [1]. While eager approaches to SMT have had some success [42], they tend to be less flexible and are therefore less frequently used.

The DPLL(T) procedure [50] [2] is a lazy approach for SMT, where the satisfiability of theory literals is checked only after a satisfying assignment at the Boolean level is found. It is a straightforward extension of the DPLL procedure, where additional interaction is provided by solvers that are specialized for particular theories (which we call a *theory solvers*). We will introduce the procedure formally in Section 4.2, and describe the basics of the procedure in this section.

Given an input problem specified by a set of clauses F, we abstract F into a purely propositional problem, by associating each theory atom in F with a corresponding propositional variable. A SAT solver determines whether a satisfying assignment exists for the abstracted problem. If such an assignment does not exist, we have determined that F is unsatisfiable. Otherwise, the solver produces a partial truth assignment M. For each theory T, a theory solver for T may either accept this assignment by determining that the set of T-literals occurring in M are consistent according to T, or reject the satisfying assignment if it is inconsistent according to T. In the former case, we have determined a theory-consistent assignment and have determined that F is satisfiable. In the latter case, we may add clauses to F that

[2]The T in the name DPLL(T) is parameterized for a fixed theory T, possibly representing a combination of theories $T_1 \cup \ldots \cup T_n$.

explain why the current state is theory-inconsistent, which may force the SAT solver to find a new assignment if these clauses are falsified by M.

When analyzing a theory inconsistent state, similar principles as in the propositional case apply, namely, we wish to add a conflict clause that will effectively rule out the current assignment M. A theory solver for T will identify a subset of the asserted literals in M that are inconsistent with T. It will then construct an explanation of the inconsistent literals in terms of a subset $\{l_1, \ldots, l_n\}$ of the literals in M, and then add $(\overline{l_1} \vee \ldots \vee \overline{l_n})$ to set of clauses F. This clause is known as a theory lemma, which we require to be a consequence of the theory T.

As mentioned, using the DPLL(T) procedure, SMT solvers capitalize on the SAT community's recent advances in performance for answering satisfiability problems at the propositional level. The performance of SMT solvers is enhanced by propagating assignments for literals that can inferred from theory reasoning. The challenge here is in calculating such propagations in an efficient manner such that the resulting performance gain outweighs this cost. Eager theory propagation can be implemented for many common SMT theories. A common example is propagating values for literals that are in the symmetric transitive closure of an equivalence relation. In other cases, such propagating disequalities in the theory of uninterpreted functions, the performance overhead typically outweighs the corresponding performance gain. In general, the developer of a theory solver in SMT can exploit this continuum, with implementations that use anything from exhaustive theory propagation to no theory propagation at all.

When checking the T-satisfiability of a formula is too expensive, the theory solver for T may require reasoning by cases. This can be accomplished within the DPLL(T) procedure by introducing additional clauses of the form $(p \vee \neg p)$ to our set F for some atom p. This technique is known as splitting on demand [4], and has been used effectively in the implementation of several methods. Termination for such approaches is guaranteed if only a finite number of such splits are requested by the solver before it arrives at a solution.

Beyond DPLL(T), other procedures exist for Satisfiability Modulo Theories [21, 45] that allow theory solvers fine-grained control over the search. In particular, this may allow a theory solver to assign values for terms in that theory, instead of relying upon propositional assignments containing these terms given by the SAT solver.

2.3 Applications

Often, SMT solvers are capable of producing useful information beyond just knowing reporting a formula is satisfiable or not. For SMT formulas, a *proof* is a trace of the reasoning used by the solver that justifies why a formula is unsatisfiable. Dually, a *model*, in its most basic form, is an assignment of values to variables which demonstrates why a formula is satisfiable.

Proofs may be useful both for the purposes of increasing the trustworthiness of the solver, and for intuition why a formula is unsatisfiable. In applications where correctness is highly critical, a proof checking procedure can be run on a generated proof of unsatisfiability for the purposes of increasing the trustworthiness of the overall system [56].

Further information can be extracted by some solvers for unsatisfiable queries. Some solvers will produce a subset of the input that is also unsatisfiable, known as a *(minimal) unsatisfiable core*. Finding an unsatisfiable core is useful for some verification applications where it is necessary to identify a small portion of a problem that is relevant in some context. Some SMT solvers are capable of generating interpolants [46, 13, 55], or formulas that summarize why a set of formulas is inconsistent with another one. Interpolants have found a wide variety of uses in recent applications, including predicate abstraction and model checking, where an over-approximation of a system's transition relation can be constructed from interpolants [44].

For satisfiable queries, model-producing SMT solvers can provide a particular valuation of the variables and non-built-in functions in a satisfiable formula, known as a model. Models are useful in a variety of formal methods applications. For software verification, a model may represent the starting state of a program for which the program exhibits an undesired behavior. By examining such a model, one can gain the intuition necessary for correcting the flaw in the program. Other uses include the synthesis of loop invariants and ranking functions [20], scheduling, and automated test case generation [29].

CHAPTER 3

HANDLING QUANTIFIED FORMULAS

In this chapter, we review various approaches for determining the satisfiability of *quantified* first-order formulas using techniques both from the fields of SMT and automated theorem proving. In particular, we will focus on instantiation-based approaches for quantified formulas.

3.1 Quantified Formulas in SMT

Many modern SMT-based applications have required the use of quantified first-order formulas for a variety of reasons. For example, quantified formulas can encode frame axioms in software verification, model run-time behaviors of certain processes, specify universal safety properties and provide axioms for a theory of interest not handled natively by the solver. For the latter, a set of quantified formulas can specify the intended interpretations for various uninterpreted symbols that represent the symbols in that theory.

While SMT solvers are known to answer quantifier-free (that is, ground) queries efficiently, their ability is limited when extended to problems with quantified formulas. The difficulty arises in DPLL(T) when checking the consistency of satisfying assignments that include universally quantified formulas, since this cannot be determined by a theory solver and moreover is undecidable in general. Although some common use cases of quantified formulas in SMT have been identified as residing within decidable fragments [28, 10], devising a general procedure is impossible.

SMT solvers must deal with both existential and universal quantification. Existentially quantified formulas are often handled by witnessing their satisfiability using a fresh set of symbols, known as *skolem symbols*. For instance, to satisfy the formula $\exists \mathbf{x}.\varphi$, where all variables in φ are in the tuple of variables $\mathbf{x}$, the SMT solver will show the satisfiability of the formula $\varphi[\mathbf{c}/\mathbf{x}]$, where the notation $[\mathbf{c}/\mathbf{x}]$ denotes replacing all occurrence of variables $\mathbf{x}$ with the skolem constants $\mathbf{c}$. Nested existentially quantified formulas can be eliminated in a similar manner, where instead a function of possibly non-zero arity is introduced. For instance, the formula $\forall x.\exists y.P(x, y)$ becomes $\forall x.P(x, f(x))$, where f is a fresh symbol. In many cases, this is performed by the solver as a preprocessing step.

For universal quantification, the underlying SAT solver will consider each universally quantified formula as a unique propositional variable. The solver will then use a scheme known as *quantifier instantiation*, in which ground instances of the quantified formula are added to the ground portion of the problem as needed. In more detail, given a formula $\forall \mathbf{x}.\varphi$, a tuple of ground terms $\mathbf{t}$ is chosen by a heuristic, and the formula $\varphi[\mathbf{t}/\mathbf{x}]$ is added to the set of formulas being considered by the solver. This may allow the solver to detect inconsistencies at the ground level, if they exist.

By applying quantifier instantiation, we can devise a sound but incomplete strategy for handling quantifiers in SMT, where a stream of ground instances of quantified formulas are considered by the solver. Such a strategy is sound since every added instance of the quantified formula is a logical consequence of the quantified formula, but it is incomplete because we are not in general guaranteed to ever find a

ground conflict and hence the process is not guaranteed to terminate. In these cases, most SMT solvers will either run indefinitely, or return an answer of "unknown" while reporting a *candidate model* to the user. A candidate model can be thought of as a structure that demonstrates the satisfiability of the current ground portion of the problem, but not necessarily the quantified portion.

An inability to answer satisfiable in the presence of quantified formulas poses a major limitation to the success of verification applications. In these applications, a satisfiable response from the SMT solver corresponds to a counterexample to a safety property that is of interest for the system in question. If the solver is unsure whether a formula is satisfiable and returns a candidate model, then the result must be manually inspected by the user to see if it is an actual counterexamples to the property in question. This is undesirable, since candidate models may have large representations that are had to inspect, and may have a high likelihood of being spurious. On the other hand, if the SMT solver can determine with confidence that a candidate model extends to the quantified portion of the problem, then the user may immediately treat it as an actual counterexample to the property in question.

In the following, current methods for handling quantifiers in SMT are reviewed, including instantiation-based methods. These methods can be summarized in their response to the following two questions:

1. Which instantiations of quantified formulas should we consider?

2. If universal quantified formulas are asserted, when can we answer satisfiable?

Much of the work on instantiation-based approaches to quantifiers in SMT has focused on the former question rather than the latter. Recent work has focused on latter question, including the approaches of model-based quantifier instantiation [28] and finite model finding [53].

3.1.1 Pattern-Based Instantiation

The most widely used and arguably most successful approach to handling quantified formulas in SMT is pattern-based quantifier instantiation, also known as E-matching [23]. This heuristic approach attempts to find ground terms that have the same shape as terms of interest for a quantified formula, and use them to guide our choice of instantiations for that formula. In more detail, for a quantified formula $\forall \mathbf{x}.\varphi$, we first find a term $f(s_1, \ldots, s_n)$ from φ that contains all of the variables $\mathbf{x}$. We will refer to $f(s_1, \ldots, s_n)$ as a *pattern* for $\forall \mathbf{x}.\varphi$. We then find a ground term $f(t_1, \ldots, t_n)$ such that $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)[\mathbf{v}/\mathbf{x}]$ are equivalent modulo a set of equalities E currently entailed by the solver. If this is the case, we use $\mathbf{v}$ when instantiating $\forall \mathbf{x}.\varphi$. Pattern-based instantiation is used primarily when determining the satisfiability of quantified formulas containing uninterpreted function symbols, that is, functions with no built-in interpretation in the theory.

Example 1 *Say we wish to determine the satisfiability of* $\forall x.f(g(x), a) \approx b$ *in the theory of equality with uninterpreted function symbols (EUF), where* $\approx$ *denotes equality. Assume that a pattern term* $f(g(x), a)$ *is provided for our formula. Say that the ground term* $f(a, a)$ *exists in our input, and that our current assignment contains the set of equalities* $\{a \approx g(c)\}$*. In this case, the pattern* $f(g(x), a)$ *matches the ground*

term $f(a, a)$ *with the substitution* $\{x \mapsto c\}$*, and as a result we will instantiate the quantified formula with* c*, giving us the instance* $f(g(c), a) \approx b$*. This may contribute to a conflict at the ground level, say in the case that the ground term* $f(a, a)$ *is disequal from* b*.*

Typically patterns for a quantified formula $\forall \mathbf{x}.\varphi$ will be terms existing in φ. As mentioned, since patterns are used to determine instantiations, we require that a pattern contain all the variables in $\mathbf{x}$. We impose additional requirements for pattern terms to control the number of instantiations produced. Most commonly, we require that a pattern have an uninterpreted function as its top-most symbol. When a quantified formula $\forall \mathbf{x}.\varphi$ does not contain a term meeting the requirements of being a pattern, we must select multiple terms $t_1, \ldots t_n$, known as a *multi-pattern*. To produce instantiations for a multi-pattern, we simultaneously match a tuple of ground term with $t_1, \ldots, t_n$ to determine instantiations for $\forall \mathbf{x}.\varphi$.

Instantiations for quantified formulas can be computed incrementally using methods known as mod-time and pattern-element optimizations [23]. These methods are capable of recognizing and constructing matches as they become feasible, such as when equalities are deduced by the solver. This may be critical to the performance of the solver, since eager approaches to quantifier instantiation have been shown to have advantages over lazy approaches [17, 27]. Additional work [17] has focused on optimizations such as calculating matches for many patterns in parallel, since this may be the performance bottleneck in problems with large amounts of ground terms.

While pattern-based quantifier instantiation is often effective at discovering conflicts, it has numerous shortcomings. First, it is difficult to judge which terms within the bodies of quantified formulas should be chosen as patterns. In fact, typical implementations will generate instantiations for *all* legal pattern terms. Many quantified formulas have a large number of terms that meet the requirements for being a pattern, including many that have no bearing on finding conflicts. This problem can be addressed by having the user provide hints as to which terms should be used as patterns. Although this solution has led to some success, it is less than ideal since the process is no longer fully automated. Moreover it makes the performance of the SMT solver highly sensitive to small modifications in patterns chosen by the user. Other techniques [32] rely on heuristics for determining which symbols are relevant for pattern selection based on the symbol's frequency in axioms.

The performance of pattern-based quantifier instantiation can be very sensitive to equivalence-preserving transformations on input formulas, since the approach depends on the syntactic structure of formulas, in particular when using patterns that contain interpreted symbols. Take for instance the pattern $x + y$. A ground term such as $5 + 1$ is a match for $x + y$, but strictly speaking 6 is not. Failure to find such matches can drastically effect the performance of the solver, even leading to a problem becoming unsolvable.

In worst-case scenarios, E-matching suffers from what is known as *matching loops*. A matching loop occurs when a repeating pattern of instantiations occur, due to terms in one iteration generating new matchable terms in the next. As an example,

consider the quantified formula $\forall x. f(x) \approx f(g(x))$, pattern $f(x)$ and ground term $f(a)$. Here, $f(a)$ is matchable with the pattern $f(x)$. Instantiating the quantified formula with a for x will produce the term $f(g(a))$. This can subsequently can be matched with $f(x)$ leading to the instantiation $g(a)$ for x and generating $f(g(g(a)))$, and so on. Some heuristics exist to avoid or explicitly break matching loops, such as keeping track of a generation level for each term we produce [27]. In these heuristics, if t is generated by instantiating a quantifier with $t_1, \ldots, t_n$, then t is given a generation level that is one higher than the maximum generation level of $t_1 \ldots t_n$. By only considering instantiations with terms having a sufficiently small generation level, we can form a strategy that essentially performs a breadth-first search for instantiations.

Even when matching loops are avoided, many similar matchable terms will be generated by quantifier instantiation, subsequently leading to an explosion in number of instantiations generated on future iterations. Depending on the form of quantifiers involved, this explosion can be quadratic or worse, particularly in the case of quantifiers that rely on multi-patterns. Even when a fair strategy is used to select instantiations, the immediate result is that the SMT solver becomes overloaded with ground clauses, making it very difficult for the solver to continue its operation.

A way to combat this explosion of terms in E-matching is to use heuristics for determining which literals contribute to the overall satisfiability of the input formula [18]. Such literals are called *relevant*, and only terms residing in such literals are considered for ground terms in the E-matching procedure. This can improve performance significantly, but does not guarantee that the number of instantiations

considered will be small, since the terms constructed as a result of quantifier instantiation may in turn end up being considered relevant.

Although less common, complications may also occur due to lack of generated instantiations. When no suitable matches can be found because of a lack of ground terms in the problem, or because our selection of pattern was too strict, the solver will produce no instantiations and return unknown. It is important to note here that the absence of matches does not imply that the problem is satisfiable. As a simple case, say our input problem is $\forall x. P(x) \wedge \forall x. \neg P(x)$. Although E-matching fails to find an instantiation, clearly this formula is unsatisfiable. Here, we must either introduce fresh ground terms arbitrarily, or relax our constraints for which patterns are usable.

Another significant disadvantage is that methods relying on pattern-based quantifier instantiation typically have no way of answering satisfiable in the presence of universally quantified formulas. Hence if the input formula is satisfiable, basic pattern-based quantifier instantiation has no hope of terminating successfully.

3.1.2 Complete Instantiation

Recent work [28] has focused on methods in SMT for answering satisfiable with input formulas containing universal quantifiers. These methods can guarantee completeness when restricted to certain decidable fragments of first-order logic with theories, such as when all variables in quantifiers are direct children of uninterpreted symbols (called the *essentially uninterpreted fragment*), and in some restricted uses of arithmetic.

In these cases, quantified formulas may be treated using a technique known

as complete instantiation. Given a set of formulas F, we first determine a *relevant domain* for each quantified formula φ in F. Based on these terms, we construct an equisatisfiable set of ground instances F^* of our formulas F. If we able to successfully determine that F^* is satisfiable, then we know that F must be satisfied by an extension of the model for F^*, which can be constructed based on the proof that F^* is equisatisfiable to F.

The calculation of the relevant domain for a particular quantified formula φ is based on existing ground terms as well as the structure of the body of φ. For example, t is in the relevant domain of function f for all ground terms $f(t)$, the relevant domain of x for a quantified formula containing the term $f(x)$ is equal to the relevant domain of f, and so on. The relevant domain of a quantified formula may be finite even in cases where the actual domain of the formula's quantifier is infinite. In such a case, it may suffice to show a quantified formula with integer variables is satisfiable by only showing that a finite set of instances F^* is satisfiable, where F^* is generated by instantiating quantified formulas in F with terms from their corresponding relevant domains. However, as mentioned, it is only possible to do so when the quantified formulas in F are of a restricted form.

3.1.3 Model-Based Quantifier Instantiation

In practice, complete instantiation is often paired with a technique known as model-based quantifier instantiation (MBQI), where we may recognize when a model for quantified formulas has been constructed without explicitly considering every instantiation occurring in F^* [28].

With MBQI, when we have a theory-consistent assignment M that satisfies all ground clauses in our problem, we construct a candidate model $\mathcal{M}$ containing interpretations for all uninterpreted function and predicate symbols in our signature. In particular, we build $\mathcal{M}$ so that it is guaranteed to satisfy all ground assertions in M. In such a candidate model, the interpretation of functions is typically given by some non-recursive lambda term of a particular form. To check whether $\mathcal{M}$ satisfies a quantified formula $\forall \mathbf{x}.\varphi$, we first replace all occurrences of uninterpreted symbols in the quantified formula φ by a term corresponding to their interpretation in $\mathcal{M}$ to obtain the formula $\varphi^{\mathcal{M}}$. Note that if all nested quantification has been removed from φ, then $\varphi^{\mathcal{M}}$ is a ground formula containing no free uninterpreted symbols. We then check the satisfiability of $\neg\varphi^{\mathcal{M}}[\mathbf{e}/\mathbf{x}]$, where $\mathbf{e}$ are fresh constants. This can be done, for instance, by invoking another instance of the SMT solver. If this formula is unsatisfiable in the background theory, then $\mathcal{M}$ is a model for the quantified formula.

Example 2 *Say we wish to determine the satisfiability of* $\{\neg P(2,3), \forall x.P(x,0)\}$. *We construct a candidate model* $\mathcal{M}$ *containing an interpretation for the predicate* P *that satisfies* $\neg P(2,3)$*, say* $P^{\mathcal{M}} := \lambda xy.\ \mathbf{false}$*, where* λ *is standard notation of function definition. To check whether* $\mathcal{M}$ *satisfies* $\forall x.P(x,0)$*, we check the satisfiability of* $\neg P^{\mathcal{M}}(e,0) = \neg(\lambda xy.\ \mathbf{false})(e,0) = \mathbf{true}$. *Since* **true** *is satisfiable, we know that* $\mathcal{M}$ *does not satisfy our quantified formula. On the other hand, say we give* P *the interpretation* $P^{\mathcal{M}} := \lambda xy.\ \neg(x \approx 2 \wedge y \approx 3)$*, or in other words* P *is only false when* x *and* y *are* 2 *and* 3 *respectively. This candidate model satisfies the quantified formula, since* $\neg P^{\mathcal{M}}(e,0) = (e \approx 2 \wedge 0 \approx 3)$ *is unsatisfiable.*

As a side effect of checking whether a model extends to a quantified formula, model-based quantifier instantiation can be used to suggest relevant instantiations. For $\forall \mathbf{x}.\varphi$, given that the check $\neg\varphi^{\mathcal{M}}[\mathbf{e}/\mathbf{x}]$ was satisfiable with the valuation $\mathbf{c}$ for $\mathbf{e}$, the solver will add $\varphi[\mathbf{c}/\mathbf{x}]$ to our set of clauses. By doing so, we are guaranteed to rule out the model $\mathcal{M}$ on future iterations. In this way, model-based quantifier instantiation can be viewed as a model refinement procedure. In the previous example, when P was given the interpretation $\lambda xy.$ **false**, say our valuation of e was 0 in the model of **true** (in this case, e could be assigned an arbitrary value). The method would subsequently instantiate the quantifier with 0 for x, adding $P(0,0)$ to our ground constraints and ruling out the model where $P^{\mathcal{M}} := \lambda xy.$ **false** on subsequent iterations, since now the definition of P must be true when x and y are 0.

Model-based quantifier instantiation is an effective method for answering satisfiable for inputs that have models where all uninterpreted functions are interpreted a certain way, such as when all integer-valued functions can be interpreted as piecewise constant over a finite number of intervals. In practice, model-based based quantifier instantiation can find helpful ground instantiations that E-matching cannot. It is often beneficial to use a combination of the two approaches, as MBQI can be used for finding unsatisfiable cases as well.

3.1.4 Quantifier Elimination

For certain classes of problems, particularly those that do not involve the use of uninterpreted functions, quantifier elimination techniques are much more effective than instantiation-based techniques, and can be used as a decision procedure for some

fragments of first-order logic. Quantifier elimination approaches determine the satisfiability of quantified formulas by building a equivalent set of quantifier-free formulas for which a decision procedure may be used. This approach can be applied to some useful cases of first-order logic, including quantified linear real arithmetic. Roughly speaking, in the linear real arithmetic case, a conjunction of ground formulas can be constructed that is equisatisfiable to quantified formula by taking relevant points based on linear inequalities occurring in a quantified formula. In this setting, alternating existential and universal quantifiers are a challenge, potentially leading to a exponential blowup in the resulting size of the generated formula. This worst-case behavior can be addressed in various ways, for instance, by tightly integrating quantifier elimination into the DPLL(T) search [7].

3.2 First-Order Theorem Proving

While the primary focus of SMT has been to efficient solve ground problems over background theories, less attention has been paid to quantified formulas. However, much research from the automated theorem proving (ATP) community has focused on quantified first-order formulas. The targeted applications of automated theorem provers typically do not involve theory reasoning, although recent research has focused on arithmetic [37, 6]. This omission is often intentional, since often decidability can be lost with the addition of background theories into first-order formulas. This section gives a brief introduction to methods used by theorem provers for handling decidable fragments of first-order logic.

In contrast to DPLL-based approaches, many classic automated theorem provers

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}\ \mathsf{Res} \qquad\qquad \frac{C \vee A \vee B}{(C \vee A)\sigma}\ \mathsf{Factor}$$
$$\text{where } \sigma = mgu(A, B). \qquad\qquad \text{where } \sigma = mgu(A, B).$$

Figure 3.1. Rules for Resolution-Based Theorem Proving

are based on the resolution calculus for pure first-order formulas in clausal form. The basic rules of the resolution calculus are shown in Figure 3.1. The resolution rule Res deduces a clause from two premises, where a formula A is in one premise, $\neg B$ is in the other, and σ is a substitution such that $A\sigma = B\sigma$. We refer to the substitution σ as the *most general unifier* (mgu) of formulas A and B. The factorization rule Factor is used to factor redundancies from a clause by finding a unifier for two of its literals.

The basic approach of resolution-based theorem proving is to apply resolution to input clauses until either (i) the empty clause is deduced, or (ii) the set of clauses becomes saturated. In the latter case, we are assured that an inconsistency cannot exist and that a model exists for our set of clauses. It can be shown that the calculus in Figure 3.1 is refutationally complete for pure first-order logic without equality. That is, if a set of clauses in pure first-order logic is unsatisfiable, then there exists a set of resolution and factoring steps for deducing the empty clause.

For equational reasoning, automated theorem provers incorporate techniques involving *paramodulation* [51]. Doing so gives solvers finer-grained control over the clauses they deduce more so than solely using resolution. The rule for paramodulation is given by the rule in Figure 3.2. Here, $D[t]_p$ represents the result of replacing the subterm of D at position p by t. This rule states that if some clause contains the

$$\frac{C \vee t \approx s \quad D}{(C \vee D[t]_p)\sigma}\ \textsf{Para}$$
where $\sigma = mgu(s, D\mid_p)$.

Figure 3.2. Rule for Paramodulation

equality $t \approx s$, and if s is unifiable with some subterm in another clause D, then we may replace that subterm with t and conclude the union of these two clauses.

Techniques involving paramodulation have been successfully integrated as part of many modern theorem provers. Some of these use a form of paramodulation known as *superposition*, in which certain term orderings are used to limit the application of paramodulation steps without loss of completeness. In both resolution and paramodulation, success is highly dependent upon controlling the number of generated clauses, as these rules naively produce far too many instances in general.

Automated theorem provers employ a number of additional optimizations for implementing calculi for first-order logic. Since many terms and clauses may exist in a problem, automated theorem provers will employ some form of *term indexing*, in which terms may be efficiently retrieved under some condition. For example, for finding resolutions of the clause $C \vee A$ for resolvent A, we are interested in quickly finding all literals $\neg B$ in our database that unify with A. Theorem provers may also contend with the large numbers of clauses produced using the aforementioned resolution calculus using redundancy criteria such as clause subsumption for identifying clauses that contain no useful information. If there exists C and C' in our set of clauses such that C' is $(C \vee D)\sigma$ for some substitution σ, then we say that C' is subsumed by C,

and we no longer need to consider it in our database of clauses. Data structures exist for recognizing these cases in an efficient manner.

3.2.1 Inst-Gen

Inst-Gen is an instantiation-based calculus [38] used by several provers in the automated theorem community. Approaches based on Inst-Gen combine propositional reasoning and instantiation in a modular fashion, taking advantage of an external SAT solver. The calculus of Inst-Gen is both sound and complete, and is also terminating for the effectively propositional reasoning (EPR) fragment, also known as the Bernays-Schonfinkel class. Problems in this class contain formulas of the form $\exists\forall\varphi$, where φ is quantifier-free and contains predicate symbols and equality but no function symbols.

In the Inst-Gen approach, all input clauses S are first instantiated with a distinguished constant $\bot$. If the resulting set of clauses $S\bot$ is unsatisfiable, then S is unsatisfiable as well and the process terminates. Otherwise, the model of the resulting set of clauses at the ground level is used to guide the instantiation process. The basic rule of Inst-Gen is shown in Figure 3.3. It varies from the rule for resolution, in that we instantiate both premises instead of performing the resolution step. The rule for Inst-Gen is clearly sound since the conclusions are instances of the premises. Furthermore, it can be shown that a set of clauses that is saturated with respect to this rule are satisfiable, given that the clauses $S\bot$ were satisfiable. Note this requires that σ is a non-empty unifier.

This approach can be enhanced by focusing on adding instantiations generated

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee A)\sigma \; (D \vee \neg B)\sigma} \; \textsf{Inst-Gen}$$
where $\sigma = mgu(A, B)$.

Figure 3.3. Rule for Inst-Gen.

only by unifying particular literals in our clauses, known as *selection literals*. The motivation here is that if clause C has at least one literal L for which $L\bot$ is true, and moreover L is not unifiable with any other literal, then we know our model extends to satisfy the clause C. Thus, the approach establishes that all clauses are satisfiable by saturating the set of clauses, using selection literals as a way to limit which instantiations are required. For example, consider the following two clauses:

Example 3 *Say we wish to determine the satisfiability of the set* $\{\forall xy.P(x,y) \vee Q(x)$, $\forall x.\neg P(x,a)\}$. *We instantiate these clauses with the distinguished representative* $\bot$ *to obtain the ground clauses* $P(\bot,\bot) \vee Q(\bot)$ *and* $\neg P(\bot,a)$. *Using a SAT solver, say we find the satisfying assignment* $\{P(\bot,\bot), \neg P(\bot,a)\}$ *for these clauses. The corresponding selection literals for our two clauses are* $P(x,y)$ *and* $\neg P(x,a)$ *which are unifiable by the substitution* $\{y \mapsto a\}$. *When applying the Inst-Gen rule, we obtain a new instance of the first clause,* $\forall x.P(x,a) \vee Q(x)$, *and add this to our clause set. Instantiating this clause with our distinguished representative, we add the ground clause* $P(\bot,a) \vee Q(\bot)$ *to the SAT solver,* $Q(\bot)$ *is assigned true and the literal* $Q(x)$ *is selected. At this point, our selection literals are* $P(x,y)$, $\neg P(x,a)$, *and* $Q(x)$, *and our clause set is saturated with respect to the Inst-Gen rule restricted to selection literals.*

In this example, since our clause set is saturated with respect to Inst-Gen, this implies a model exists, in this case the one where P is true except when its second argument is equal to a, and Q is always true. Each quantified formula is satisfied by virtue of the first-order extension of its selection literal, noting that all relevant instances have been processed since our clause set is saturated.

Recent work has extended this approach to handle equational reasoning [36]. However, incorporating additional theories remains a challenge.

3.2.2 Finite Model Finding

While highly studied techniques exist for resolution-based theorem proving, other research has focused on finding models for non-theorems. The following section covers standard techniques used for finding finite models of first-order formulas.

3.2.2.1 MACE-Style Model Finding

The most widely-used approach for model finding for first-order formulas is the MACE-style approach [43], named for the MACE tool. In this approach, the question of whether a first-order formula has a model of a fixed (finite) size can be encoded as a satisfiability problem in propositional logic. Model finders that use this approach will incrementally fix domain sizes and use an underlying SAT solver to search for finite models.

To convert a first-order formula to propositional logic, function applications $f(x)$ and equality can be eliminated from a problem by introducing a set of propositional symbols representing when $f(x)$ is equivalent to an element in our domain. The semantics of the function f can be preserved by imposing constraints on these

symbols, namely its totality and that $f(x)$ can only be equal to one value. For instance, if we are searching for models with domain size 2, for a 0-ary function g, we introduce the propositional variables g_1 and g_2 representing the cases that g is the first and second element of our domain respectively, and impose that $(g_1 \vee g_2)$ enforcing the totality of g and $(\neg g_1 \vee \neg g_2)$ enforcing that g only returns one value. Given a model can be found for the resultant propositional encoding of a problem, then a model can be constructed for the original problem.

This form of model finding relies on an exhaustive instantiation of non-ground (quantified) clauses. In general, a clause $C[x_1, \ldots, x_n]$ for domain size of k is converted into k^n ground clauses representing all instances of that clause. Thus, a downside of a MACE-style approach is that a larger overhead may be incurred when converting the problem to this form. Notice that introducing variables into first-order clauses by techniques such as term flattening lead to an exponential increase in the number of instances we need to check. To combat these problems, competitive implementations are enhanced by a variety of methods including static symmetry reduction, clause splitting, and sort inference.

Static symmetry reduction seeks to find additional constraints that can be added to the solver for a problem, while preserving its satisfiability [16]. For example, say we are searching for a model of size k for a set of clauses S containing the terms $t_1 \ldots t_n$. In the standard approach, this is done by introducing a set of k distinct constants $c_1 \ldots c_k$ representing the elements of our domain. Using symmetry reduction techniques, we may assume t_1 is c_1, t_2 is either c_1 or c_2, and so on without loss of

generality. Such techniques have been shown to lead to significant performance in MACE-style approaches, and are applicable to other classes of problems as well.

Clause splitting can be used to reduce the maximum number of variables for clauses in our problem [16]. We may identify cases when a clause $C[X] \cup D[Y]$ can be converted into the clauses $\{S(X \cap Y)\} \cup C[X]$ and $\{S(X \cap Y)\} \cup D[Y]$ where S is a fresh predicate symbol, given some criteria for when such a split is possible. The maximum number of variables in the resulting two clauses may be less than the original clause, potentially reducing the number of required instantiations by an order of magnitude.

Sort inference is another technique used for improving performance in this setting [16]. In sort inference, we compute constraints concerning the sorts of terms in our signature. For example, if our problem contains the equality $f(t) = u$, then we know that the sort of u and $f(t)$ must be the same, and that the argument sort of f must be equal to the sort of t. From these constraints, we may effectively treat certain sets of terms as having distinct sorts. This leads to improved performance, since symmetry reduction can be applied to these sets independently, and hence the solver can effectively assume various terms are equal without loss of generality. Additionally, if we are able to infer that variables can be treated as having a sort with a smaller domain, then we can reduce the instances of clauses we need to consider.

3.2.2.2 SEM Model Finding

The SEM [61] approach to model finding differs from the approach of MACE in that it does not convert the problem to propositional logic. The technique has built-

in treatment for equality, and uses constraint propagation techniques that resemble modern SMT solvers. Symmetry reduction is incorporated using heuristic techniques including a least-number heuristic, which enables the solver to avoid searching for isomorphic models. While MACE-style methods are generally more widely-used than SEM-style methods, a number of model finders have been influenced by the approach of SEM. In practice, SEM-style methods are effective for handling problems that involve equational reasoning.

CHAPTER 4

FORMAL PRELIMINARIES

In this section, we formally introduce definitions used in the remainder of the thesis, and the DPLL(T) procedure used by modern SMT solvers.

4.1 Preliminaries

We work in the context of many-sorted first-order logic with equality. A (many-sorted) *signature* Σ consists of a set of sort symbols and a set of *(sorted) function symbols*, $f : S_1 \times \cdots \times S_n \to S$, where $n \geq 0$ and $S_1, \ldots, S_n, S$ are sorts in Σ. When n is 0, f is also called a *constant symbol*. We use the binary predicate $\approx$ to denote equality. We assume that Σ includes a Boolean sort Bool and constants **true** and **false** of that sort—allowing us to encode all other predicate symbols as function symbols of return sort Bool. For each sort S, we assume our signature contains the if-then-else function symbol $ite : \mathsf{Bool} \times S \times S \to S$.

Given a signature Σ, a Σ-term is either a variable x, or an expression of the form $f(t_1, \ldots, t_n)$, where f is a function from Σ, and $t_1, \ldots, t_n$ are Σ-terms. A term t is a *well-sorted* term of sort S if t is a variable having sort S, or t is of the form $f(t_1, \ldots, t_n)$ where f has rank $S_1 \times \cdots \times S_n \to S$, and $t_1, \ldots, t_n$ are well-sorted terms of sorts $S_1, \ldots, S_n$ respectively. An atomic Σ-formula is an equality $t_1 \approx t_2$ where t_1 and t_2 are well-sorted terms of the same sort. A Σ-literal is either an atomic Σ-formula p or its negation $\neg p$. A Σ-clause is a disjunction of Σ-literals, e.g. $l_1 \vee \ldots \vee l_n$. We will use the symbol $\perp$ to denote an empty disjunction of literals. A Σ-formula is an

expression built from atomic Σ-formulas, and logical connectives such as $\vee$, $\wedge$, and $\neg$. A *ground term (resp. formula)* is a Σ-term (resp. formula) with no variables. An occurrence of variable x is *free* in a formula φ if it does not reside within a subformula $\forall x.\psi$ or $\exists x.\psi$ of φ. We write $FV(\varphi)$ to denote the set of occurrences of variables that are free in φ, or the *free variables* of φ. A Σ-*sentence* is a Σ-formula with no free variables. Where $\mathbf{x} = (x_1, \ldots, x_n)$ is tuple of sorted variables we write $\forall \mathbf{x}\, \varphi$ as an abbreviation of $\forall x_1 \cdots \forall x_n\, \varphi$. A Σ-formula is *universal* if it has the form $\forall \mathbf{x}\, \varphi$ where φ is a quantifier-free formula.

A substitution σ is a mapping from variables to terms of the same sort, such that the set $\{x \mid x\sigma \neq x\}$, the *domain* of σ (written $\mathcal{D}om(\sigma)$), is finite. We say σ is a *grounding substitution* (for $\mathbf{x}$) if σ maps each variable in $\mathbf{x}$ to a ground term.

A Σ-*structure* $\mathcal{M}$ maps each sort S in Σ to a non-empty set $S^{\mathcal{M}}$, the *domain* of S in $\mathcal{M}$, and each function symbol $f : S_1 \times \cdots \times S_n \rightarrow S \in \Sigma$ to a total function $f^{\mathcal{M}} : S_1^{\mathcal{M}} \times \cdots \times S_n^{\mathcal{M}} \rightarrow S^{\mathcal{M}}$. The evaluation of a term $f(t_1, \ldots, t_n)$ in $\mathcal{M}$, denoted $M[\![t]\!]$ is defined recursively, such that $\mathcal{M}[\![f(t_1, \ldots, t_n)]\!] = f^{\mathcal{M}}(\mathcal{M}[\![t_1]\!], \ldots \mathcal{M}[\![t_n]\!])$. The evaluation of an if-then-else term $ite(\varphi, t_1, t_2)$ is defined such that $\mathcal{M}[\![ite(\varphi, t_1, t_2)]\!] = \mathcal{M}[\![t_1]\!]$ if $\mathcal{M}[\![\varphi]\!] = \mathcal{M}[\![\mathbf{true}]\!]$, and $\mathcal{M}[\![t_2]\!]$ otherwise. For Σ-structure $\mathcal{M}$ and a substitution σ mapping variables to elements of its domain in $\mathcal{M}$, we write $\mathcal{M}\sigma$ to denote a structure interpreting a term t as $x\sigma$, if t is a variable x in the domain of σ, and $\mathcal{M}[\![t]\!]$ otherwise. A satisfiability relation $\models$ between Σ-structures and Σ-sentences, written $\models$, is defined as follows.

- $\mathcal{M} \models t_1 \approx t_2$ iff $\mathcal{M}[\![t_1]\!] = \mathcal{M}[\![t_2]\!]$

- $\mathcal{M} \models \varphi \wedge \psi$ iff $\mathcal{M} \models \varphi$ and $\mathcal{M} \models \psi$

- $\mathcal{M} \models \varphi \vee \psi$ iff $\mathcal{M} \models \varphi$ or $\mathcal{M} \models \psi$

- $\mathcal{M} \models \neg\varphi$ iff $\mathcal{M} \not\models \varphi$

- $\mathcal{M} \models \forall \mathbf{x}.\varphi$ iff $\mathcal{M}\sigma \models \varphi$ for all grounding substitutions σ for $\mathbf{x}$

A Σ-structure $\mathcal{M}$ *satisfies* (or *is a model of*) a Σ-sentence φ if $\mathcal{M} \models \varphi$. A formula is satisfiable if and only if it has a model.

A *theory* given a signature Σ is defined as a set of deductively closed Σ-formulas [1] We call function symbols occurring in T as *interpreted*, and all other function symbols as uninterpreted. A formula φ is satisfiable modulo theory T if and only there exists a model satisfying φ that also satisfies T. A set Γ of formulas T-*entails* a Σ-formula φ, written $\Gamma \models_T \varphi$, if every model of T that satisfies all formulas in Γ satisfies φ as well. We say that a set of formulas Γ *propositionally entails* a formula φ, written $\Gamma \models_p \varphi$, if the set $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable. when considering all atomic formulas in it as propositional variables.

Let F be a set of ground clauses, and let M be a satisfiable set of literals that propositionally entails F. We will refer to M as a *satisfying assignment* for F. Let $\mathbf{T}_M$ be the set of all terms occurring in M. A set $E \subseteq \{s \approx t \mid s, t \in \mathbf{T}_M\}$ is a *congruence (for M)* if it is closed under entailment: for all $s, t \in \mathbf{T}_M$, $E \models s \approx t$ iff $s \approx t \in E$. The *congruence closure* E^* *of* E *with respect to* M is the smallest

[1] For simplicity, we define a theory here as a set of Σ-formulas Ax. More strictly, a theory is defined as a set of (intended) interpretations that satisfy Ax.

congruence for M that includes E. By construction, E^* is an equivalence relation over $\mathbf{T}_M$. It can be shown (see, e.g., [2]) that E is satisfied by a structure $\mathcal{M}$ that interprets each sort S as $\mathbf{V}^S = \{v_1^S, \ldots, v_{n_S}^S\}$ consisting of an arbitrary representatives for each equivalence class of E^* over terms of sort S. We will call $\mathcal{M}$ a *normal model.*

Given a normal model $\mathcal{M}$, a *model assignment* is a pair, written $t \mapsto v$, where t is a term and v is a value from the domain of $\mathcal{M}$. Given a congruence closure E^* for M with representatives $\mathbf{V}^S$ for sort S, we may construct a set of model assignments $\mathcal{A}_M$ consisting of $t \mapsto v_i$ for all $t \in \mathbf{T}_M$, where v_i is the representative term in the equivalence class of E^* containing t. We will call $\mathcal{A}_M$ an *evaluation map* for M. We will write $\mathcal{A}_M(t)$ to denote the value that t is mapped to in $\mathcal{A}_M$.

4.2 DPLL(T) Procedure

This section formally presents the DPLL(T) procedure, which determines the T-satisfiability of a ground set of clauses for a background theory T. In this section and in the remainder of the thesis, we will consider a theory $T = T_1 \cup \ldots \cup T_m$ where each T_i is a theory of signature Σ_i. We call *free* those sort and function symbols whose interpretation is not restricted in any way by any of the theories, and consider them as part of the EUF signature; we call *built-in* all the others. For convenience and without loss of generality, we assume that $\Sigma_1, \ldots, \Sigma_m$ have the same set $\mathbf{S}$ of sort symbols (including the Boolean sort Bool), and share a distinguished finite set $\mathcal{C}_S$ of free constants of sort S for each $S \in \mathbf{S}$. Let $\mathcal{C} = \bigcup_{S \in \mathbf{S}} \mathcal{C}_S$. We impose the restriction that the signatures $\Sigma_1, \ldots, \Sigma_m$ share no function symbols, besides the constants in $\mathcal{C}$.

We describe the DPLL(T) procedure for the theory T as a state transition

system. States are triples of the form $\langle M, F, C\rangle$ where

- M, the current *assignment*, is a sequence of literals and *decision points* $\bullet$,
- F is a set of ground clauses derived from the original input problem, and
- C is either the distinguished value no or a clause, which we will refer to as a *conflict clause*.

Each assignment M can be factored uniquely into the subsequence concatenation $M_0 \bullet M_1 \bullet \cdots \bullet M_n$, where no M_i contains decision points. For $i = 0, \ldots, n$, we call M_i the *decision level* i of M and denote with $M^{[i]}$ the subsequence $M_0 \bullet \cdots \bullet M_i$. When convenient, we will treat M as the set of its literals and call them the *asserted literals*. The formulas in F have a particular *purified form*[2] that can be assumed with no loss of generality since any formula can be efficiently converted into that form while preserving satisfiability in T: each element of F is a ground clause, and each atom occurring in F is *pure*, that is, has signature Σ_i for some $i \in \{1, \ldots, m\}$.

Initial states have the form $\langle \emptyset, F_0, \mathsf{no}\rangle$ where F_0 is an input set of formulas to be checked for satisfiability. The expected final states are $\langle M, F, \bot\rangle$ (which we will call a *fail state*), when F_0 is unsatisfiable in T; or $\langle M, F, \mathsf{no}\rangle$ with M satisfiable in T, F equisatisfiable with F_0 in T (that is, F is satisfiable if and only if F_0 is satisfiable), and $M \models_{\mathrm{p}} F$.

Transition rules The possible behaviors of the system are defined by a set of non-deterministic state transition rules, specifying a set of successor states for each current

[2]For details, see Appendix A.

$$\textsf{\textbf{Propagate}}_i \quad \frac{l_1, \ldots, l_n \in \mathsf{M} \quad l_1, \ldots, l_n \models_i l \quad l \in \mathrm{L}_{\mathsf{F}} \cup \mathrm{I}_{\mathsf{M}} \quad l, \bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M}\; l}$$

$$\textsf{\textbf{Decide}} \quad \frac{l \in \mathrm{L}_{\mathsf{F}} \cup \mathrm{I}_{\mathsf{M}} \quad l, \bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M} \bullet l} \qquad \textsf{\textbf{Conflict}}_i \quad \frac{\mathsf{C} = \mathsf{no} \quad l_1, \ldots, l_n \in \mathsf{M} \quad l_1, \ldots, l_n \models_i \bot}{\mathsf{C} := \bar{l}_1 \vee \cdots \vee \bar{l}_n}$$

$$\textsf{\textbf{Explain}}_i \quad \frac{\mathsf{C} = l \vee D \quad \bar{l}_1, \ldots, \bar{l}_n \models_i \bar{l} \quad \bar{l}_1, \ldots, \bar{l}_n \prec_{\mathsf{M}} \bar{l}}{\mathsf{C} := l_1 \vee \cdots \vee l_n \vee D} \qquad \textsf{\textbf{Learn}} \quad \frac{\mathsf{C} \neq \mathsf{no}}{\mathsf{F} := \mathsf{F} \cup \{\mathsf{C}\}}$$

$$\textsf{\textbf{Learn}}_i \quad \frac{\emptyset \models_i l_1 \vee \cdots \vee l_n \quad l_1, \ldots, l_n \in \mathrm{L}_{\mathsf{M}}|_i \cup \mathrm{I}_{\mathsf{M}} \cup L_i}{\mathsf{F} := \mathsf{F} \cup \{l_1 \vee \cdots \vee l_n\}}$$

$$\textsf{\textbf{Backjump}} \quad \frac{\mathsf{C} = l_1 \vee \cdots \vee l_n \vee l \quad \mathsf{lev}\; \bar{l}_1, \ldots, \mathsf{lev}\; \bar{l}_n \leq \; i < \mathsf{lev}\; \bar{l}}{\mathsf{C} := \mathsf{no} \quad \mathsf{M} := \mathsf{M}^{[i]}\; l}$$

Figure 4.1. DPLL($T_1, \ldots, T_m$) rules

state.[3] The rules are provided in Figure 4.1 in *guarded assignment form* [41]. A rule applies to a state s if all of its premises hold for s. In the rules, M, F and C respectively denote the assignment, formula set, and conflict clause component of the current state. The conclusion describes how each component is changed, if at all. We write $\bar{l}$ to denote the complement of literal l and $l \prec_{\mathsf{M}} l'$ to indicate that l occurs before l' in M. The function lev maps each literal of M to the (unique) decision level at which l occurs in M. The set L_{F} (resp., L_{M}) consists of all ground literals in F (resp., all literals of M) and their complements. For $i = 1, \ldots, m$, the set $\mathrm{L}_{\mathsf{M}}|_i$ consists of the Σ_i-literals of L_{M}. I_{M} is the set of all *interface literals* of M: the equalities and disequalities between constants c, d with c and d occurring in $\mathrm{L}_{\mathsf{M}}|_i$ and $\mathrm{L}_{\mathsf{M}}|_j$ for two distinct $i, j \in \{1, \ldots, m\}$.

[3]To simplify the presentation, we do not consider here rules that model the forgetting of learned lemmas and restarts of the SMT solver.

The index i ranges from 0 to m for the rules **Propagate**$_i$, **Conflict**$_i$ and **Explain**$_i$, and from 1 to m for **Learn**$_i$. In all rules, $\models_i$ abbreviates $\models_{T_i}$ when $i > 0$. In **Propagate**$_0$, $l_1, \ldots, l_n \models_0 l$ simply means that $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee l \in \mathsf{F}$. Similarly, in **Conflict**$_0$, $l_1, \ldots, l_n \models_0 \bot$ means that $\bar{l}_1 \vee \cdots \vee \bar{l}_n \in \mathsf{F}$; in **Explain**$_0$, $\bar{l}_1, \ldots, \bar{l}_n \models_0 \bar{l}$ means that $l_1 \vee \cdots \vee l_n \vee \bar{l} \in \mathsf{F}$. The rules **Decide**, **Propagate**$_0$, **Explain**$_0$, **Conflict**$_0$, **Learn**, and **Backjump** model the behavior of the SAT engine, which treats ground atoms as Boolean variables. The rules **Conflict**$_0$ and **Explain**$_0$ model the conflict discovery and analysis mechanism used by CDCL SAT solvers.

All the other rules model the interaction between the SAT engine and the individual theory solvers in the overall SMT solver. Generally speaking, the system uses the SAT engine to construct the assignment M as if the problem were propositional, but it periodically asks the sub-solvers for each theory T_i to check if the set of Σ_i-constraints in M is unsatisfiable in T_i, or entails some yet undetermined literal from $\mathrm{L}_{\mathsf{F}} \cup \mathrm{I}_{\mathsf{M}}$. In the first case, the sub-solver returns an *explanation* of the unsatisfiability as a conflict clause, which is modeled by **Conflict**$_i$ with $i = 1, \ldots, m$. The propagation of entailed theory literals and the extension of the conflict analysis mechanism to them is modeled by the rules **Propagate**$_i$ and **Explain**$_i$. The inclusion of the interface literals I_{M} in **Decide** and **Propagate**$_i$ achieves the effect of the Nelson-Oppen combination method [58, 12]. The rule **Learn**$_i$ is needed to model theory solvers following the splitting-on-demand paradigm [4]. When asked about the satisfiability of their constraints, these solvers may instead return a *splitting lemma*, a formula valid in their theory and encoding a guess that needs to be made about the

constraints before the solver can determine their satisfiability. The set L_i in the rule is a finite set consisting of literals, not present in the original formula F_0, which may be generated by such solvers.

Executions and correctness An *execution* of a transition system modeled as above is a (possibly infinite) sequence $s_0, s_1, \ldots$ of states such that s_0 is an initial state and for all $i \geq 0$, s_{i+1} can be generated from s_i by the application of one of the transition rules. A system state is *irreducible* if no transition rules besides $\mathsf{\textbf{Learn}}_i$ apply to it. An *exhausted execution* is a finite execution whose last state is irreducible. A *complete execution* is either an exhausted execution or an infinite execution. An application of $\mathsf{\textbf{Learn}}_i$ is *redundant* in an execution if the execution contains a previous application of $\mathsf{\textbf{Learn}}_i$ with the same premise.

Adapting results from [50, 41, 4], it can be shown that every execution ending in $\langle M, F, C\rangle$ satisfies the following invariants: M contains only pure literals and no repetitions; $F \models_T C$ and $M \models_{\mathrm{p}} \neg C$ when $C \neq \mathsf{no}$; every model of T satisfying F satisfies the initial set of formulas. Moreover, the transition system is *terminating*: every execution with no redundant applications of $\mathsf{\textbf{Learn}}_i$ is finite; and *sound*: for every execution starting with a state $\langle \emptyset, F_0, \mathsf{no}\rangle$ and ending with $\langle M, F, \bot\rangle$, the clause set F_0 is unsatisfiable in T. Under suitable assumptions on the sub-theories $T_1, \ldots, T_m$, the system is also *complete*: for every exhausted execution starting with $\langle \emptyset, F_0, \mathsf{no}\rangle$ and ending with $\langle M, F, \mathsf{no}\rangle$, M is satisfiable in T and $M \models_{\mathrm{p}} F_0$. Here, we provide a sketch of the correctness proof for DPLL$(T_1, \ldots, T_m)$.

Theorem 1 *DPLL*$(T_1, \ldots, T_m)$ *is sound, terminating, and complete for every set of*

ground clauses F_0*.*

Proof: (Sketch) To show soundness, notice that all reachable states of the form $\langle M, F, C\rangle$ where $C \neq \mathsf{no}$ are such that C is T-entailed by F. When applying **Conflict**$_i$, either C is T_i-entailed by some theory $i > 0$, or is a clause from F. When applying **Explain**$_i$, we replace a literal l in C with a set of literals $l_1, \ldots, l_n$, which are entailed by l either according to the theory when $i > 0$, or together with F when $i = 0$. Thus, when a state of the form $\langle M, F, \bot\rangle$ is reachable, then $\bot$ is entailed by F, and since all clauses added to F are entailed by F_0, $\bot$ is entailed by F_0 as well.

To show termination, notice that the set of literals $\mathrm{L}_{\mathsf{M}} \cup \mathrm{I}_{\mathsf{M}} \cup L_1 \cup \ldots \cup L_m$ is finite, and for all reachable states $\langle M, F, C\rangle$, we have that every literal in M and occurring in clauses from F and C belong to this set. As a consequence, only a finite number of states exist, and **Learn** and **Learn**$_i$ can only be applied a finite number of times. Consider a partial ordering $\succeq$ on assignments M, with maximal element $\emptyset$, such that $(e_1 M_1) \succeq (e_2 M_2)$ if either $e_1 = \bullet \neq e_2$, or $e_1 = e_2$ and $M_1 \succeq M_2$. In addition, consider a partial ordering $\succeq$ on conflict clauses such that $C_1 \succeq C_2$ if either C_1 is no, or C_1 and C_2 are not no and $C_2 \prec_M^{mul} C_1$, where $\prec_M^{mul}$ is the multiset order for $\prec_M$. Extend this ordering to states such that $\langle M_1, F_1, C_1\rangle \succeq \langle M_2, F_2, C_2\rangle$ if and only if $M_1 \succeq M_2$ or $M_1 = M_2$ and $C_1 \succeq C_2$. Applying all other rules (besides **Learn** and **Learn**$_i$) to state s result in state s' where $s' \prec s$. Since a finite number of states exist, the procedure terminates.

To show completeness, notice that if we are in an irreducible state $\langle M, F, \mathsf{no}\rangle$, then M is a T-consistent satisfying assignment for F. To see this, since **Decide** does

```
proc check(M, F, C) ≡
  (Propagate_0 | ... | Propagate_n)*;
  if weak_effort(M, F, C)
    if ∃l ∈ L_F. l, l̄ ∉ M
      Decide on l
    else if strong_effort(M, F, C)
      return ⟨M, F, no⟩
    end
  end
  return check_conflict(M, F, C)
```

```
proc check_conflict(M, F, C) ≡
  if C ≠ no
    (Explain_0 | ... | Explain_n)*;
    if C = ∅
      return ⟨M, F, ⊥⟩
    else
      Learn; Backjump
    end
  end
  return check(M, F, C)
```

Figure 4.2. A typical strategy check for applying DPLL($T_1, \ldots, T_m$) rules. In this method, weak_effort and strong_effort (not given) apply possibly multiple applications of **Learn**$_1$ ... **Learn**$_n$, or one application of **Conflict**$_i$ for some $1 \leq i \leq n$. These methods return false only when they apply at least one rule; strong_effort(M, F, C) returns true only when M is consistent according to $T_1 \cup \ldots \cup T_m$.

not apply, M must contain an assignment for all literals in F and moreover is a satisfying assignment for F since **Conflict**$_0$ does not apply. Since **Conflict**$_i$ does not apply for any $i > 0$, then M must be consistent according to T. Since $F_0 \subseteq F$, we have that M propositionally entails our input F_0. Assuming we are given complete procedures for determining when **Conflict**$_i$ applies, and a sufficient strategy for propagating a set of interface literals in I_M to ensure combined satisfiability between theories, we are guaranteed that M (and thus F_0) is satisfiable according to $T_1 \cup \ldots \cup T_m$. Thus, since our procedure is terminating, it is also complete. ∎

4.2.1 A Typical Strategy for DPLL(T)

Figure 4.2 gives a typical strategy for applying the rules of DPLL(T). Given a state $\langle M, F, C \rangle$, we apply the procedure check(M, F, C), whose pre-condition is that C is no. We first apply the rule **Propagate**$_i$ for sub-theories T_i, possibly multiple times. Afterwards, we apply a *weak effort check*, as given by the subprocedure weak_effort,

which will either apply **Conflict**$_i$ for some $1 \leq i \leq n$, or otherwise may apply **Learn**$_i$ possibly multiple times. Frequent weak effort checks are commonly used in SMT solvers [50]. They are useful during the extension of the assignment L, to avoid extensions that are clearly unsatisfiable in one of the theories.

When no conflicts or clauses are learned at weak effort, we apply **Decide** on some unassigned literal l from L_F, if one exists. Otherwise, our assignment M is complete, and we apply a *strong effort check*, where the solver may apply **Conflict**$_i$ or **Learn**$_i$ possibly multiple times, and return false. Otherwise, it will return true if it can determine that M is satisfiable in T, after which the method returns the (final) state $\langle M, F, \mathsf{no} \rangle$, indicating that F is satisfiable.

When a conflict clause C is discovered, either at weak or strong effort, we may perform conflict analysis by repeated applications of **Explain**$_i$. If we reach a state of the form $\langle M, F, \perp \rangle$, then we know F is unsatisfiable, and we return. Otherwise, we may add a learned clause via **Learn**, and apply **Backjump** to return to a previous part of the search.

For soundness and termination, we require that each call to weak_effort and strong_effort legally executes a set of DPLL($T_1, \ldots, T_m$) rules each of which are not redundant in the current execution. For termination, we require that weak_effort and strong_effort return false only when they apply at least one rule. For completeness, we require that strong_effort returns true only when M is satisfiable in $T_1 \cup \ldots \cup T_m$.

CHAPTER 5

FINITE MODEL FINDING IN SMT

Finite model finding techniques can be used in the context of SMT for answering satisfiable in the presence of universally quantified formulas [53]. Recall that SMT solvers work with sorted logics containing both interpreted and uninterpreted sorts. Finite model finding focuses on finding *finite models*, that is, models that interpret each uninterpreted sort as a finite set. The approach mentioned in this chapter will be applicable to cases where each universal quantifier in the problem is either over an uninterpreted sort, or a finite interpreted sort. Examples of interpreted finite sorts include fixed length bit-vectors, finite (non-recursive) datatypes, as well as certain cases of integer arithmetic where bounds on the quantifiers are explicitly provided or can be inferred. Many applications of SMT rely on problems that fall into such categories, given a careful encoding of the constraints they require.

An approach for finite model finding in SMT has advantages over both existing approaches for quantifiers in SMT, as well as standard approaches to finite model finding used by automated theorem provers. For the first, most algorithms used by SMT solvers focus largely on finding proofs of unsatisfiability when quantified formulas are asserted, answering unknown when a proof cannot be found. Enabling SMT solvers to answer satisfiable for many problems containing quantifiers fills a significant need for the automated reasoning community. In particular, when an SMT solver fails to find a proof, it can provide a concrete counterexample. While model

finding methods for first-order formulas are well developed by the ATP community, finite model finding in SMT has the advantage that (ground) decision procedures for background theories can be combined modularly with techniques for reasoning about quantified formulas. In industrial applications, the use of background theories is nearly always required.

5.1 A Model-Based Approach for Quantifiers in SMT

In the following, we give an overview of our approach for finite model finding in SMT, given an input formula ψ. Like standard approaches to handling quantifiers in SMT, we first perform preprocessing steps to convert ψ into a purified form, as described in Appendix A. As a result of this step, ψ is converted into a set of clauses F_0, where each clause in F_0 is either ground, or an equivalence of the form $a \Leftrightarrow \forall \mathbf{x}\, \varphi$, where a is a Boolean variable, and each other occurrence of a in F_0 has positive polarity. In this procedure, a will serve as a proxy for $\forall \mathbf{x}\, \varphi$ at the ground level.

Say we have converted ψ to a set of clauses F_0 of this form. In the remainder of this chapter, we will assume our input has been converted in this way. Our approach will be based on constructing satisfying assignments M for an evolving set of ground clauses F, where $F_0 \subseteq F$. We will say a universally quantified formula $\forall \mathbf{x}\, \varphi$ is *active* in M if and only if $a \in M$, where $a \Leftrightarrow \forall \mathbf{x}\, \varphi \in F$. The approach is parameterized by a *quantifier instantiation heuristic* $\mathcal{H}$, which adds instances of active quantified formulas to F.

Definition 1 (Model Finding Procedure)

1. *Find a satisfying assignment M for F. Otherwise, if none exists, return "unsatisfiable".*

2. *From M, construct a candidate model $\mathcal{M}$ that satisfies F. Let $\mathbf{V}$ be the union of the domain elements of each sort S in $\mathcal{M}$.*

3. *Let Q be the set of quantified formulas that are active in M. Using $\mathcal{H}$, for each $\forall \mathbf{x}\, \varphi \in Q$ where $a \Leftrightarrow \forall \mathbf{x}\, \varphi \in F$, choose a set $I_{\mathbf{x}}$ of substitutions from $\mathbf{x}$ to terms in $\mathbf{V}$, and add the instances $\{\neg a \vee \varphi\sigma \mid \sigma \in I_{\mathbf{x}}\}$ to F. If the union of all these sets is empty, return "satisfiable", otherwise go to Step 1.*

In Step 3, we assume preprocessing techniques from Appendix A are applied to each $\neg a \vee \varphi\sigma$, and thus each instance may correspond to multiple clauses that are added to F.

The following sections will examine these steps in more detail. In Section 5.2, we describe strategies for finding satisfying assignments to the ground set of clauses F (Step 1). In particular, we will focus on finding satisfying assignments that induce candidate models with a small number of domain elements $\mathbf{V}$. In Section 5.3, we describe various ways for representing and constructing candidate models $\mathcal{M}$ (Step 2). In Section 5.4, we describe methods for checking the satisfiability of our set of active quantified formulas Q based on $\mathcal{M}$, and various strategies for choosing the

substitutions $I_{\mathbf{x}}$ (Step 3).

5.2 EUF with Finite Cardinality Constraints (EFCC)

In this section, we introduce the theory of EUF with finite cardinality constraints (EFCC), which will be used as a way of finding candidate models with small domain sizes. We describe its signature (Σ_{EFCC}), give a decision procedure for a conjunction of ground constraints in this theory, and describe how it can be integrated into the DPLL($T_1, \ldots, T_m$) architecture as described in Section 4.2. We then describe an efficient solver for EFCC that works well in practice, and how it can be used for minimizing the number of equivalence classes in the congruence closure maintained by the solver. Finally, we show a strategy, *fixed-cardinality DPLL*($T_1, \ldots, T_m$), for establishing finite cardinality bounds for uninterpreted sorts during the DPLL($T_1, \ldots, T_m$) procedure.

Definition 2 (Theory EFCC **of** EUF **with finite cardinality constraints)** *The signature* Σ_{EFCC} *of* EFCC *extends the signature of* EUF *with a constant* $\mathsf{card}_{S,k}$ *of sort* Bool *for each free sort* S *and integer* $k > 0$*. Its models are all* Σ_{EFCC}*-interpretations that satisfy each* $\mathsf{card}_{S,k}$ *exactly when they interpret* S *as a set of cardinality* $n \leq k$*.*

Note that the only ground atoms in EFCC besides those of the form $\mathsf{card}_{S,k}$ are equalities. It is not difficult to show, using reductions to and from graph coloring, that the satisfiability of ground literals in EFCC is an NP-complete problem.

5.2.1 Decision Procedure

This section presents a decision procedure for determining the satisfiability of ground constraints with the theory of EUF with finite cardinality constraints. For now, we assume a signature Σ_{EFCC} containing a single uninterpreted sort S. As input, our procedure takes a set M consisting of cardinality constraint literals for S, and equalities and disequalities over ground Σ_{EFCC}-terms of sort S.

Decision Procedure for EFCC**:** First, construct a congruence closure E^* over the terms of sort S from M. If there exists a $t \approx s \in E^*$ for which $t \not\approx s \in M$, return unsatisfiable. If there exists no positively asserted cardinality literal in M, return satisfiable. Otherwise, assume k is the least integer such that $\mathsf{card}_{S,k} \in M$. If there exists $\neg\mathsf{card}_{S,j}$ in M, where $j \geq k$, return unsatisfiable. If there are k or fewer equivalence classes in E^*, return satisfiable. If there exists two equivalence classes in E^* with representatives s and t such that $M \not\models s \not\approx t$, split the problem into $M \cup s \approx t$ and $M \cup s \not\approx t$, that is, return satisfiable if either branch is satisfiable, and unsatisfiable otherwise. Otherwise, there are $k+1$ equivalence classes with representatives $t_1, \ldots, t_{k+1}$ where $M \models t_i \not\approx t_j$ for all $1 \leq i \neq j \leq k+1$; report unsatisfiable. □

Lemma 1 *The above procedure is sound, terminating and complete for every set of ground* Σ_{EFCC}*-literals* M*.*

Proof: To show that the procedure is sound, first note that splitting the problem based on equalities $s \approx t$ is sound, since all models satisfy exactly one of $s \approx t$

and $s \not\approx t$. We either answer unsatisfiable when an equality $t \approx s$ is entailed by M where $t \not\approx s$ is also in M, when conflicting literals $\mathsf{card}_{S,k}$ and $\neg\mathsf{card}_{S,j}$ are asserted for $j \geq k$, or when $k+1$ equivalence classes exist containing terms that are entailed to be disequal by M. For conflicts of the second type, no model can be constructed containing less than or equal to k and more than j elements in the domain of S. For conflicts of the third type, no model can be constructed satisfying $\mathsf{card}_{S,k}$, since more than k equivalence classes are entailed to be disequal in M.

To show termination, assuming that no conflict is found between cardinality literals, we will construct a congruence closure E^*, consisting of a set of equivalence classes with a set of representatives, call them $\mathbf{V}^S$. If successful, it can be shown that splitting on the equality of s and t decreases the size of the set $\{(s,t) \mid s,t \in \mathbf{V}^S, s \neq t, M \not\models s \not\approx t\}$, or in other words, the number of equivalence classes that are pairwise not entailed to be disequal. In either branch when splitting on $s \approx t$, no equivalence classes are created (although two existing ones are possibly merged), and (s,t) is no longer an element of this set. When this set is empty, the algorithm is guaranteed to terminate, since either more than k equivalence classes are entailed to be distinct, in which case the procedure answers unsatisfiable, or otherwise there are k or fewer equivalence classes, in which case the procedure answers satisfiable.

To show completeness, we answer satisfiable when our congruence closure E^* contains no equality whose negation occurs in M, and either no cardinality literal is asserted positively in M, or E^* contains less than or equal to k equivalence classes where k is the smallest integer such that $\mathsf{card}_{S,k} \in M$. In either case, we may construct

a model where S is interpreted as a set of size j, and $j \leq k$ for all $\mathsf{card}_{S,k} \in M$, and $j \geq k$ for all $\neg\mathsf{card}_{S,k} \in M$. If j is greater than the number of equivalence classes in E^*, new elements can be added to the domain of S without affecting the satisfiability of the equalities and disequalities in M. Thus, since the procedure is terminating, it is also complete. ■

Corollary 1 *Every satisfiable ground set of* Σ_{EFCC}*-literals has a finite model.*

Proof: Whenever our decision procedure answers satisfiable, we may construct a finite model where S is interpreted as the set of representative terms from each equivalence class, as well as a (finite) number of additional elements that ensure that all literals of the form $\neg\mathsf{card}_{S,j}$ in M are satisfied. ■

5.2.2 Integration into DPLL($T_1, \ldots, T_m$)

We now describe how the decision procedure for EFCC can be integrated into the DPLL($T_1, \ldots, T_m$) framework, where T_i is EFCC. While updating the current state $\langle M, F, C\rangle$, we assume a standard algorithm for computing a congruence closure for M and for reporting conflicts when none exists.

In addition to determining M is unsatisfiable due to congruence, the decision procedure reports "unsatisfiable" in one of two ways, each of which corresponds to a legal application of **Conflict**$_i$. First, when M contains conflicting cardinality literals for some sort S, we may apply **Conflict**$_i$ with $\mathsf{C} := (\neg\mathsf{card}_{S,k} \vee \mathsf{card}_{S,j})$ for $j > k$. Second, when M contains $\mathsf{card}_{S,k}$ and entails that there are more than k equivalence classes of sort S, we may apply **Conflict**$_i$ with $\mathsf{C} := (\overline{l_1} \vee \ldots \vee \overline{l_n} \vee \neg\mathsf{card}_{S,k})$, where

$l_1, \ldots, l_n$ is a set of equalities and disequalities from M that entail that some terms $t_1, \ldots, t_n$ are distinct in all models of M. In either case, the negated conjunction of the literals from the conflict imply a contradiction.

To model the splitting as performed by the decision procedure, when M contains $\mathsf{card}_{S,k}$ and more than k equivalence classes currently exist, we may apply the rule $\textbf{Learn}_i$ to add $(s \approx t \vee s \not\approx t)$ to our set of clauses F for two terms s, t residing in two different equivalence classes where $M \not\models s \not\approx t$. Since the algorithm for constructing a congruence closure will not introduce new terms beyond those in F_0, we are ensured that s and t are terms that exist in our original set of clauses F_0.

Theorem 2 *$DPLL(T_1, \ldots, T_m, T_{\mathrm{EFCC}})$ is sound, terminating and complete for every set of ground clauses F_0.*

Proof: Following the requirements from Section 4.2, the procedure is sound since we apply $\textbf{Conflict}_i$ only to clauses whose negated literals imply a contradiction and $\textbf{Learn}_i$ to clauses that hold in all models. To show our procedure is terminating, the only literals introduced by applications of $\textbf{Learn}_i$ (call them L_{EFCC}) are equalities and disequalities between terms occurring in F_0. Since the set L_{EFCC} is finite, by the same argument used in Theorem 1, the procedure terminates. We can show completeness using the same argument as Lemma 1. In particular, the procedure is terminating, and only terminates in a final state $\langle M, F, \mathsf{no} \rangle$ when the congruence closure we constructed contains k or fewer equivalence classes for all $\mathsf{card}_{S,k} \in M$. In such states, we are guaranteed that M is satisfiable in EFCC. ■

5.2.3 Efficient Solver

We now describe techniques that make our solver for EFCC efficient in practice when integrated in the DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) architecture. In the following, we describe the operation of the solver during weak and strong effort checks (as introduced in Section 4.2.1) given a satisfying assignment M. For now, we assume that M contains constraints for a single uninterpreted sort S only. We also assume that at least one cardinality literal for S has been asserted positively in M (otherwise, the solver for EFCC acts similar to a standard solver for EUF), and that k is the smallest integer such that $\mathsf{card}_{S,k} \in M$.

5.2.3.1 Weak Effort Check

At weak effort, we recognize conflicting states of three different forms, as mentioned in Section 5.2.1. First, if we are unable to construct a congruence closure for M that is consistent with the disequalities from M, we report a conflict that describes the inconsistency. Second, given $\mathsf{card}_{S,k} \in M$, if $\neg\mathsf{card}_{S,j}$ is asserted in M for $j > k$, we report the conflict $(\neg\mathsf{card}_{S,k} \vee \mathsf{card}_{S,j})$. Third, we may recognize cases when there are $k + 1$ equivalence classes $t_1, \ldots, t_{k+1}$ that are currently entailed to be disequal. In this case we use $\mathbf{Learn}_i$ to add the lemma $(\neg\mathsf{distinct}(t_1, \ldots t_{k+1}) \vee \neg\mathsf{card}_{S,k})$ to F, where $\mathsf{distinct}(t_1, \ldots t_{k+1})$ is shorthand for the conjunction of disequalities stating that the terms $t_1, \ldots, t_{k+1}$ are pairwise distinct elements. We will refer to the aforementioned lemma as a *clique lemma*. As mentioned earlier, we could alternatively apply $\mathbf{Conflict}_i$ to report a conflict of form $(\overline{l_1} \vee \ldots \vee \overline{l_n} \vee \neg\mathsf{card}_{S,k})$, where $l_1, \ldots, l_n$ are equalities and disequalities that entail $\mathsf{distinct}(t_1, \ldots t_{k+1})$. However, we have found

that in practice that this is inefficient, as many different sets of literals can be found for essentially the same conflict.

For the purposes of discovering unsatisfiable states of the third form, our procedure will incrementally maintain a *disequality graph* D for S, whose vertices correspond to the equivalence classes of sort S, and whose edges represent disequalities between these equivalence classes. When convenient, we will identify these equivalence classes with their representative terms. In this representation, a sufficient condition for discovering a conflict reduces to finding a $(k+1)$-clique in the disequality graph D. Now, even just checking for the presence of a $(k+1)$-clique in a n-vertex graph is too expensive in general—as its worst-case complexity is $O(n^{k+1}(k+1)^2)$. For this reason, our procedure will be based on an incomplete check for candidate cliques. This is done by partitioning the vertices of the graph into *regions*.

Definition 3 (k-Region) *Given an undirected graph $D = (V, E)$ and a set of vertices $R \subseteq V$. For a vertex $v \in R$, let $\mathsf{ext}(v)$ be the number of edges between v and vertices not in R. We say R is a k-*region *of D if for all $0 \leq i \leq k$, the size of the set $\{v \mid v \in R, \mathsf{ext}(v) \geq i\}$ is less than or equal to $k - i$. A k-*regionalization *$\mathcal{R}_D$ of D is a partition of V into k-regions, which we will refer to as simply a regionalization when k is understood or not important.*

Lemma 2 *If $\mathcal{R}_D$ is a k-regionalization of D, and D contains a k-clique C, then all the vertices in C reside in the same region of $\mathcal{R}_D$.*

Proof: If $k \leq 1$, the statement is trivial. Otherwise, assume by contradiction D contains k-clique $C = C_1 \cup C_2$ for non-empty C_1, C_2, where $v \in R$ for each $v \in C_1$

and $v \notin R$ for each $v \in C_2$, where R is some region of $\mathcal{R}_D$. Say $\mid C_2 \mid= i$, and thus $\mid C_1 \mid= k-i$. Since C is a k-clique, we have that $\mathsf{ext}(v)$ must be greater than or equal to i, for all $v \in C_1$, contradicting the assumption that R is a region. ■

Notice that any graph $D = (V, E)$ has a trivial regionalization, which partitions V into one set containing all vertices in V.

Example 4 *Consider the constraints* $\{c_1 \not\approx c_2, c_2 \not\approx c_3, c_3 \not\approx c_4\}$, *all over sort* S, *and the partition* $\{\{c_1, c_2\}, \{c_3, c_4\}\}$. *This partition is a 3-regionalization in the disequality graph induced by this set, because a 3-clique can span two regions only if it contains two vertices with interregional edges, and this partition only has one such edge. Adding the disequality* $c_2 \not\approx c_4$ *or* $c_1 \not\approx c_4$ *breaks the regionalization invariant.*

Let us examine how to maintain a k-regionalization in an (initially empty) *evolving graph* D, that is, one supporting the dynamic allocation of vertices and edges, as well as the merge operation on vertices. These operations will be triggered by operations performed on the congruence closure data structure maintained by the solver. Assuming we have a regionalization $\mathcal{R}_D$ for graph D, we show how to construct a regionalization $\mathcal{R}_{D'}$ for the resulting graph D' obtained as a result of each of these operations. In the following, $\mathcal{R}(v)$ denotes the region in a regionalization $\mathcal{R}$ that contains the vertex v.

Adding Vertices When a vertex v is added to D, $\mathcal{R}_{D'}$ is the result of adding the singleton region $\{v\}$ to $\mathcal{R}_D$. □

Adding Edges When we add an edge (v_1, v_2) to D, we have that $\mathcal{R}_{D'} = \mathcal{R}_D$ is still

```
proc fix_region(R, 𝓡) ≡
  if R is not a k-region
    choose some R′ ∈ 𝓡, where R′ ≠ R
    𝓡 := 𝓡 \ {R, R′} ∪ {R ∪ R′}
    fix_region({R ∪ R′}, 𝓡)
  end
```

Figure 5.1. The fix_region procedure. This procedure is called on region R in k-regionalization $\mathcal{R}$. As a heuristic, we choose the R' with the highest density of interregional edges to R.

a partition of V. However, $\mathcal{R}_D(v_1)$ or $\mathcal{R}_D(v_2)$ may not be regions of D'. We apply the procedure fix_region from Figure 5.1 first to $(\mathcal{R}_D(v_1), \mathcal{R}_{D'})$ and then to $(\mathcal{R}_D(v_2), \mathcal{R}_{D'})$ to ensure that $\mathcal{R}_{D'}$ is a regionalization. □

Merging Vertices When a vertex v_1 is merged with another vertex v_2 in D, we have that D' is a quotient graph of D, that is, D' contains a new vertex, call it u, connected to all vertices that are connected to either v_1 or v_2 in D. If $\mathcal{R}_D(v_1)$ is equal to $\mathcal{R}_D(v_2)$, let R be $(\mathcal{R}_D(v_1) \cup \{u\}) \setminus \{v_1, v_2\}$. Then $\mathcal{R}_{D'}$ is equal to $(\mathcal{R}_D \cup R) \setminus \{\mathcal{R}_D(v_1)\}$. To ensure $\mathcal{R}_{D'}$ is a regionalization, we apply fix_region to $(R, \mathcal{R}_{D'})$. If $\mathcal{R}_D(v_1)$ is not equal to $\mathcal{R}_D(v_2)$, let $\{v_i, v_j\} = \{v_1, v_2\}$, $R_i = (\mathcal{R}_D(v_i) \cup \{u\}) \setminus \{v_i\}$, and $R_j = \mathcal{R}_D(v_j) \setminus \{v_j\}$. Then, $\mathcal{R}_{D'}$ is equal to $(\mathcal{R}_D \cup \{R_i, R_j\}) \setminus \{\mathcal{R}_D(v_1), \mathcal{R}_D(v_2)\}$. We apply fix_region to $(R_i, \mathcal{R}_{D'})$ and subsequently to $(R_j, \mathcal{R}_{D'})$. □

Given that $\mathsf{card}_{S,k}$ is asserted in M, we are interested in finding cliques of size $k+1$ in the disequality graph D for S induced by M. For this purpose, our solver maintains a $(k+1)$-regionalization $\mathcal{R}_D$ of D. We will call each region with at least $k+1$ vertices a *large region*, and all others *small regions*. For the purposes of

efficiently discovering $k+1$-cliques, we will maintain a *watched set* of $k+1$ vertices for each large region R in $\mathcal{R}_D$, which we will write as $w(R)$. This set is maintained incrementally when vertices are added or removed from regions, and when regions are combined.

Maintaining watched sets of vertices helps recognize conflicting states during a weak effort check. If there exists a large region R in $\mathcal{R}_D$ where each vertex in $w(R)$ is connected, then we add the clique lemma $(\neg\mathsf{distinct}(t_1, \ldots, t_{k+1}) \vee \neg\mathsf{card}_{S,k})$ to F using the rule $\textbf{Learn}_i$, where $w(R) = \{t_1, \ldots, t_{k+1}\}$.

5.2.3.2 Strong Effort Check

During a strong effort check, our solver must determine that the current set of constraints is consistent, or otherwise report a conflict or lemma. As mentioned in Section 5.2.1, our solver does the former only when k or fewer equivalence classes of sort S exist. Otherwise, we will choose two equivalence classes that are not currently entailed to be distinct to identify. This choice is guided the watched set of vertices within regions. In particular, for each large region R in $\mathcal{R}_D$, we know that $w(R)$ does not form a clique. We choose two vertices $t_i, t_j \in w(R)$ that are not connected in D, and use $\textbf{Learn}_i$ to add the lemma $(t_i \approx t_j \vee t_i \not\approx t_j)$ to F. We also tell the solver that it should subsequently decide on $t_i \approx t_j$ with positive polarity. Otherwise, if no large regions exist in $\mathcal{R}_D$, then either D contains fewer than $k+1$ vertices, in which case we may answer "satisfiable", or otherwise there must exist at least two small regions. In the latter case, we select two regions R_i and R_j based on a heuristic[1], combine

[1]Namely, the maximum density of interregional edges.

them into a new region $R_i \cup R_j$, apply $\mathsf{fix_region}$ to $R_i \cup R_j$, and repeat the strong effort check.

We illustrate the operation of the EFCC solver with a couple of examples.

Example 5 *Consider the constraints* $\{a \approx f(b),\ b \approx f(c),\ a \not\approx b,\ b \not\approx c,\ \mathsf{card}_{S,2}\}$ *where all terms are over the single sort S. First, the* EFCC *solver computes the congruence* $\{\{a, f(b)\},\ \{b, f(c)\},\ \{c\}\}$*. Using a, b, c as the representatives, the solver builds the disequality graph with edges* $\{(a, b), (b, c)\}$*. Since* $\mathsf{card}_{S,2}$ *limits the size of S to at most 2, the solver generates the lemma* $a \approx c \lor a \not\approx c$*. Adding the constraint* $a \approx c$ *produces no conflicts and allows the* EFCC *solver to answer "satisfiable".*

Example 6 *Consider the constraints* $\{c_1 \approx c,\ c_4 \approx c,\ c_1 \not\approx c_2,\ c_2 \not\approx c_3,\ c_3 \not\approx c_4,\ \mathsf{card}_{S,2}\}$ *where all the constants have sort S. The corresponding disequality graph for these constraints contains a clique of size 3. By discovering that clique, the* EFCC *solver can conclude that it is impossible to shrink the model to 2 elements, and hence reports a clique lemma of the form* $\neg\mathsf{distinct}(c_1, c_2, c_3) \lor \neg\mathsf{card}_{S,2}$*.*

Because of congruence constraints, guesses on merge lemmas may sometimes lead to inconsistencies when constructing the congruence closure, unless we compute and propagate all entailed disequalities—which is usually not the case, for efficiency reasons. This is demonstrated in the following example.

Example 7 *Consider the constraints* $\{c_3 \approx f(c_1),\ c_4 \approx f(c_2),\ c_3 \not\approx c_4,\ \mathsf{card}_{S,2}\}$ *where all the terms have sort S. Unless the solver propagates the entailed literal* $c_1 \not\approx c_2$*, the* EFCC *solver will construct the disequality graph* $(\{c_1, c_2, c_3, c_4\}, \{(c_3, c_4)\})$ *for S, and*

may decide to assert $c_1 \approx c_2$*. The subset* $\{c_3 \approx f(c_1),\ c_4 \approx f(c_2), c_3 \not\approx c_4,\ c_1 \approx c_2\}$ *of the new assignment will then be found unsatisfiable by congruence closure. In contrast, deciding* $c_1 \approx c_3$ *and* $c_2 \approx c_4$ *will produce a model of the required cardinality.*

It is immediate that the solver in this section described in this section is also sound. To argue that it is terminating, notice that fix_region is terminating since each recursive call to this procedure reduces the number of regions in $\mathcal{R}_D$ by one, and similarly for repeated calls of the strong effort check. Additionally, we have that all introduced literals (either those when reporting clique lemmas at weak effort, or when splitting on equalities at strong effort) are taken from the finite set of equalities and disequalities between terms occurring in our original clause set F_0. Since the conditions for answering satisfiable are the same as those as mentioned in Lemma 1 and the solver mentioned in this section is terminating, it is also complete.

5.2.4 Establishing Finite Cardinalities

We have now shown that an efficient solver for EFCC can be integrated into the DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) architecture. Now, we focus our attention on how the EFCC theory solver can be used for finding a satisfying assignment for a set of ground clauses F containing a bounded number of equivalence classes of a particular sort S. In other words, given a satisfiable set of ground clauses F, our procedure will find a (ideally minimal) integer $k > 0$ such that $F \wedge \mathsf{card}_{S,k}$ is satisfiable. Due to Corollary 1, one trivial way to do this would be the following. First, use the solver to determine if $F \wedge \mathsf{card}_{S,1}$ is satisfiable, and answer satisfiable if so. If this is unsatisfiable, use the solver to determine if $F \wedge \mathsf{card}_{S,2}$ is satisfiable, and so on. An immediate disadvantage

of this approach is that in the absence of conflict analysis, it will not be able to determine that F is unsatisfiable. In contrast, the following approach does not have this restriction.

We assume the use of the EFCC theory solver as described in the previous section, which is based on the strategy for DPLL($T_1, \ldots, T_m$) as introduced in Section 4.2.1. We impose the following modifications. When the weak effort check does not produce a conflict, we find the least integer $k > 0$ such that $\neg \mathsf{card}_{S,k}$ is not asserted in M. If $\mathsf{card}_{S,k}$ is not in M, we ensure that $\mathsf{card}_{S,k}$ is a literal in F by applying $\textbf{Learn}_i$ with $(\mathsf{card}_{S,k} \vee \neg \mathsf{card}_{S,k})$. Then, when choosing a decision literal, we insist that **Decide** be applied to $\mathsf{card}_{S,k}$.

In this approach, all other applications of **Decide** occur when $\mathsf{card}_{S,k} \in M$ for some k. In other words, all search is performed for a fixed cardinality for S. For this reason, we call the approach mentioned here *fixed-cardinality DPLL*($T_1, \ldots, T_m, T_{\mathrm{EFCC}}$).

5.2.4.1 Extension to Multiple Sorts

Now, let us consider the case when our signature Σ contains multiple sorts $S_1, \ldots, S_n$, and we want to establish finite cardinalities for each of them. Given a set of input clauses F, our goal will be to determine that either F is unsatisfiable, or find a tuple $(k_1, \ldots, k_n)$ such that $F \wedge \mathsf{card}_{S_1,k_1} \wedge \ldots \wedge \mathsf{card}_{S_n,k_n}$ is satisfiable. One challenge is having a *fair* strategy when considering which cardinalities to increment in the case of conflicts. For instance, consider the formula $(c \not\approx d \vee \varphi)$, where c and d are constants of sort S_1, and the formula φ entails that no finite models of sort S_2

exist [2]. Clearly this formula has a model where the cardinality of sorts S_1 and S_2 are $(2, 1)$ respectively. However, in the absence of a fair strategy, a naive approach could search for models of size $(1, 1)$, $(1, 2)$, $(1, 3)$, and so on, ad infinitum.

To extend fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) so that it is fair in the presence of multiple sorts, we extend the signature Σ of EFCC to include Boolean constants of the form $\mathsf{card}_{\Sigma,k}$ for each integer $k > 0$. A Σ-interpretation $\mathcal{I}$ that interprets each sort $S_i \in \Sigma$ as a set of size k_i for $1 \leq i \leq n$, satisfies $\mathsf{card}_{\Sigma,k}$ if and only if $k_1 + \ldots + k_n \leq k$.

We then use a two-tiered strategy for choosing decision literals. First, if no literal of the form $\mathsf{card}_{\Sigma,k}$ exists in M, we decide on $\mathsf{card}_{\Sigma,k}$ for the least $k \geq n$ such that $\neg\mathsf{card}_{\Sigma,k} \notin M$, ensuring that this literal exists in F using $\mathbf{Learn}_i$. Second, we determine if $\mathsf{card}_{\Sigma,k}$ is in conflict with the negated cardinality literals from sorts $S_1, \ldots, S_n$. If so, we report a conflict of the form $(\mathsf{card}_{S_{i1},k_{i1}} \vee \ldots \vee \mathsf{card}_{S_{im},k_{im}} \vee \neg\mathsf{card}_{\Sigma,k})$, where $i_1, \ldots, i_m$ are distinct, and $k_{i1} + \ldots + k_{im} + n > k$. In the case when no conflict of this form is found, if no literal of the form card_{S_i,k_i} exists in M, we similarly apply $\mathbf{Learn}_i$ (if necessary) and subsequently decide on card_{S_i,k_i} for a sort S_i, where k_i is the least $k_i \geq 1$ such that $\neg\mathsf{card}_{S_i,k_i} \notin M$. The order on which we decide cardinality literals between sorts is arbitrary. In the presentation, we will assume we choose cardinality literals for S_i before S_j where $i < j$.

For consistency, we assume that even in the case Σ contains a single unin-

[2] In reality, φ must contain quantifiers for this to be the case. As a result, fairness will not be an issue until later sections where quantified formulas are addressed.

terpreted sort, we use the methods introduced for ensuring fairness between multiple sorts. The following summarizes the invariant maintained by fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$), in particular, that it decides upon (minimal) cardinality literals positively before deciding upon any other literal.

Proposition 1 *Given a signature Σ containing uninterpreted sorts $S_1, \ldots, S_n$, for each execution of fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) ending in $\langle M, F, C\rangle$, either M contains no decision points, or M is of the form $N \bullet \mathsf{card}_{\Sigma,k}\ M_0\ \ (\bullet \mathsf{card}_{S_1,k_1} M_1) \cdots\ \ (\bullet \mathsf{card}_{S_m,k_m}\ M_m)\ N'$, for some m, $0 \leq m \leq n$, where $N, M_0, M_1, \ldots, M_m$ contain no decision points, N' contains no decision points if $m < n$, $\neg\mathsf{card}_{\Sigma,j} \prec_M \mathsf{card}_{\Sigma,k}$ for each $n \leq j < k$, and $\neg\mathsf{card}_{S_i,j} \prec_M \mathsf{card}_{S_i,k_i}$ for each $1 \leq i \leq m$, $1 \leq j < k_i$.*

The following lemma states that fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) eventually rules out the current cardinality k of sorts in our signature whenever no model of that size exists. The procedure will either find a conflict that depends on the cardinality constraint, or otherwise find a conflict that is independent of cardinality, thereby showing the input is unsatisfiable.

Lemma 3 *Given a set of ground clauses F_0, if $F_0 \wedge \mathsf{card}_{\Sigma,k}$ is unsatisfiable, then every complete execution of fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) for F_0 having a prefix ending in $\langle M \bullet \mathsf{card}_{\Sigma,k}, F, \mathsf{no}\rangle$ also has a prefix ending in either a fail state, or in $\langle M\ \ N\neg\mathsf{card}_{\Sigma,k}, F', \mathsf{no}\rangle$, where N contains no decision points, and $F \subseteq F'$.*

Proof: Assume we have an execution e of fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) that has a finite prefix ending in state $\langle M \bullet \mathsf{card}_{\Sigma,k}, F, \mathsf{no}\rangle$ for which the lemma does not

hold. Since fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) is complete due to Theorem 2, because of the invariant stated in Proposition 1, e must be an infinite execution where $\neg\mathsf{card}_{\Sigma,k}$ is not in the assignment of any state in e. In all such states, the literals L_{EFCC} introduced by applications of **Learn**$_i$ consist (at most) of the set of all equalities and disequalities between terms from F, and literals of the form $\mathsf{card}_{S_i,j}$ and $\neg\mathsf{card}_{S_i,j}$ for $j < (k - n)$ for each sort S_i in Σ. Since this set is finite, using a similar argument as the one for termination in Theorem 1, we have that e cannot be infinite, thus contradicting our assumption. ■

Theorem 3 *Fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) is sound, terminating and complete for every set of ground clauses F.*

Proof: Assume our signature Σ contains uninterpreted sorts $S_1, \ldots, S_n$. The procedure is sound as a result of Theorem 2, and that splitting on cardinality literals preserves all models.

To show it is terminating, we consider two cases. In the case that F is unsatisfiable, for each $1 \leq i \leq n$, let k_i be smallest integer greater than or equal to the number of terms of sort S_i in F, and such that the literal $\neg\mathsf{card}_{S_i,k_i}$ does not occur in F. Let k be the smallest integer greater than or equal to $k_1 + \ldots + k_n$, and such that the literal $\neg\mathsf{card}_{\Sigma,k}$ does not occur in F. Assume that there exists an infinite execution e of fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$). As a consequence of Lemma 3 for cardinalities $n, \ldots, k$, we can conclude there is a prefix of e ending in $\langle M \neg\mathsf{card}_{\Sigma,k}, F', \mathsf{no}\rangle$. For this to be the case, either $\neg\mathsf{card}_{\Sigma,k}$ was added to the assignment due to propagation or due to **Backjump**. In either case, we have that $\neg\mathsf{card}_{\Sigma,k}$

either must exist as a literal in some clause in F', or must be implied by a set of literals from our assignment. Due to our selection of k, this is a contradiction. Thus, when F is unsatisfiable, every execution of fixed-cardinality DPLL$(T_1, \ldots, T_m, T_{\text{EFCC}})$ for F terminates.

In the case that F is satisfiable, since it is ground, due to Corollary 1, it must have a model with domain size k_i for S_i for some finite integer $k_i > 0$, for each $1 \leq i \leq n$. Let $k = k_1 + \ldots + k_n$. It can be shown that the set of literals L_{EFCC} introduced by applications of $\textbf{Learn}_i$ in this procedure consists (at most) of the set of all equalities and disequalities between terms from F, literals of the form $\mathsf{card}_{\Sigma,j}$ and $\neg\mathsf{card}_{\Sigma,j}$ for $j = n, \ldots, k$, as well as all literals of the form $\mathsf{card}_{S_i,j}$ and $\neg\mathsf{card}_{S_i,j}$ for $j < (k - n)$, for each sort S_i. Since this set is finite, by the same argument as Theorem 1, the procedure terminates, and thus for the same argument as in Theorem 2, it is also complete. ■

5.3 Constructing Candidate Models

Now that we have seen how satisfying assignments M are constructed for a ground set of clauses F, we will focus our attention to constructing candidate models $\mathcal{M}$. Assuming a signature Σ, a candidate model $\mathcal{M}$ is a Σ-structure that satisfies F, and may also satisfy our set of (active) quantified formulas Q as well.

We construct a candidate model $\mathcal{M}$ containing a finite set of domain elements $\mathbf{V}^S$ for each sort S occurring in Σ, and complete definitions for all function and predicate symbols of Σ. Our model construction uses a particular choice of domain elements, which will ensure the finite model completeness and refutational complete-

ness of our approach in the presence of quantified formulas. In our construction, the definition a function symbol f of sort $S_1 \times \ldots \times S_n \rightarrow S$, is conceptually a (finite) map from $\mathbf{V}^{S1} \times \ldots \mathbf{V}^{Sn}$ to $\mathbf{V}^S$. We will construct the interpretation of each function symbol f from the set of asserted ground equalities in M with terms containing f. Notice that collecting those equalities typically produces only a partial definition for f. To complete the interpretation of f, one can use arbitrary values for the missing function tuples. We will see various strategies for choosing these values in this section.

5.3.1 Choosing Domain Elements

As mentioned, for each uninterpreted sort S in our signature, the domain elements $\mathbf{V}^S$ in a candidate model $\mathcal{M}$ can be taken from the congruence closure E^* for M. For the sake of showing termination in the presence of our quantified formulas in Section 5.5, we will impose additional restrictions on which terms can be chosen for $\mathbf{V}^S$, based on the following definition.

Definition 4 *The* depth *of a ground term t, written* $\mathsf{depth}(t)$*, is defined inductively such that* $\mathsf{depth}(f(t_1, \ldots, t_n)) = 1 + \mathsf{max}(\{\mathsf{depth}(t_i) \mid 1 \leq i \leq n\} \cup \{0\})$.

When choosing a representative term for an equivalence class in E^* containing term t, we choose a term s of minimal depth such that $E^* \models t \approx s$, where s may or may not be a term occurring in E^*. Such a term can be found using the following algorithm. First, for every equivalence class containing some constant term c, we choose c as the representative term for that equivalence class. Let E_0 be the set of equivalence classes containing a constant. For $n > 0$, let E_n be the set containing E_{n-1}, as well as all

equivalence classes containing a term t such that $E^* \models f(t_1, \ldots, t_n) \approx t$, and $t_1, \ldots, t_n$ occur in E_{n-1}. In this case, we assign $f(t_1, \ldots, t_n)$ as the representative term of the equivalence class containing t.

Example 8 *Say we are given a congruence closure E^* with equivalence classes $\{a, g(a)\}$ and $\{f(g(a))\}$. Our algorithm will choose a as the representative of the first equivalence class, and subsequently choose $f(a)$ for the second equivalence class, noting that $E^* \models f(a) \approx f(g(a))$.*

It is easy to see this algorithm eventually assigns terms to all equivalence classes, and that $\mathsf{depth}(t) \leq\mid \mathbf{V}^S \mid$ for all representative terms t of sort S. To show the latter, notice that each equivalence class occurring in E_i but not E_{i-1} is assigned a representative term having depth of exactly i. Assuming that there is a representative term having depth j for some $j >\mid \mathbf{V}^S \mid$ means that we computed the sets $E_0 \subsetneq \ldots \subsetneq E_j$. However, this is impossible because the size of E_j is at most $\mid \mathbf{V}^S \mid$.

The above strategy is enough to ensure the finite-model completeness of our approach on inputs where quantification is limited to uninterpreted sorts. To show our approach is refutationally complete, we will require that for each equivalence class in E^* containing term t, we choose a representative s where $\mathcal{M}[\![s]\!] = \mathcal{M}[\![t]\!]$, and s has minimal depth with respect to all such Σ-terms. Unfortunately, to find s, we must first construct $\mathcal{M}$, and thus our definition is circular. To solve this, we can assume $\mathcal{M}$ is first constructed using the techniques mentioned throughout this section, and

afterwards the representative term for t is replaced with such an s. [3]

5.3.2 Representing Function Definitions

Now that we have seen how domain elements are chosen for a candidate model $\mathcal{M}$, we focus on constructing representations for function symbols. Such functions will be represented using the data structures described in this section. In the following, we use symbols such as v or w to refer to values, that is, the domain elements $\mathbf{V}$ of $\mathcal{M}$. We use symbols u to refer to *abstract values*, where:

$$u := \bot \mid v \mid * \tag{5.1}$$

Conceptually, $\bot$ is intended to mean no value, and $*$ is any value. The *concretization* of an abstract value for uninterpreted term t, written $\gamma(u)[t]$, is a formula where:

$$\gamma(\bot)[t] := \mathbf{false} \tag{5.2}$$

$$\gamma(v)[t] := (t \approx v) \tag{5.3}$$

$$\gamma(*)[t] := \mathbf{true} \tag{5.4}$$

An abstract value u has sort S if u is $*$, $\bot$, or is a value v of sort S. For $\gamma(u)[t]$ where t is a term of sort S, we require u to have sort S as well. We order abstract values according to a partial ordering $\succeq$, such $u \succeq u'$ if and only if $\gamma(u')[x] \Rightarrow \gamma(u)[x]$ is valid in the theory of equality for a variable x. We will say u *generalizes* u' if $u \succeq u'$. We say u and u' are *compatible* if and only if $\gamma(u)[x] \wedge \gamma(u')[x]$ is satisfiable in the

[3] While doing so allows us to show the refutational completeness of our approach, for performance reasons (see Section 5.7), this strategy is not used in our implementation.

theory of equality. Define the *meet* of two abstract values, written $u \bigtriangleup u'$, such that $\gamma(u \bigtriangleup u')[x]$ is equivalent to $\gamma(u)[x] \wedge \gamma(u')[x]$ in the theory of equality. It can be shown that the meet of two abstract values is well defined, i.e. $u \bigtriangleup u' = \bot$ if either u or u' is $\bot$, $u \bigtriangleup * = u$, $* \bigtriangleup u' = u'$, and $v_1 \bigtriangleup v_2 = v_1$ if $v_1 = v_2$ and $\bot$ otherwise. We extend these notions to n-tuples of abstract values, written as symbols such as $\mathbf{c}$ and $\mathbf{d}$, which we call *conditions*. We will commonly write $\mathbf{c}.i$ to denote the ith element of tuple $\mathbf{c}$.

The concretization of condition $\mathbf{c}$ for uninterpreted terms $\mathbf{t}$ is defined as:

$$\gamma(\mathbf{c})[\mathbf{t}] := \gamma(\mathbf{c}.1)[\mathbf{t}.1] \wedge \ldots \wedge \gamma(\mathbf{c}.n)[\mathbf{t}.n] \tag{5.5}$$

Two conditions are *compatible* if $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{x}]$ is satisfiable in the theory of equality, where $\mathbf{x}$ is a tuple of distinct variables. The *meet* of two conditions $\mathbf{c} \bigtriangleup \mathbf{d}$ is defined such that $\gamma(\mathbf{c} \bigtriangleup \mathbf{d})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{x}]$ in the theory of equality. Any tuple containing $\bot$ as an element we will write simply as $\bot$. We call $\bot$ the empty condition and all others non-empty. Extend the partial order $\succeq$ to conditions, such that $\mathbf{c} \succeq \mathbf{d}$ if and only if $\gamma(\mathbf{d})[\mathbf{x}] \Rightarrow \gamma(\mathbf{c})[\mathbf{x}]$ is a tautology in the theory of equality. The condition $\mathbf{c} \bigtriangleup_i u$ refers to $\mathbf{c}$ with the element at position i replaced by the element $\mathbf{c}.i \bigtriangleup u$. For example, $(*, v_2) \bigtriangleup_1 v_1 = (v_1, v_2)$.

The following data structure can be used for representing the interpretation of functions in a candidate model.

Definition 5 (Function Definition) *A* definition *of arity n is a list of entries $\mathbf{c}_1 \rightarrow t_1, \ldots, \mathbf{c}_m \rightarrow t_m$, where $t_1 \ldots t_m$ are terms, $\mathbf{c}_1 \ldots \mathbf{c}_m$ are non-empty n-tuples of abstract values, and $\mathbf{c}_i \not\succeq \mathbf{c}_j$ for all $1 \leq i < j \leq m$.*

We will write $\emptyset$ to denote the definition containing no entries. A definition is *complete* if and only if it contains an entry of the form $(*, \ldots, *) \to t$, which we will write simply as $* \to t$. The *concretization* of a complete definition is defined as follows, where *ite* is the logical if-then-else operator:

$$\gamma(* \to t) := t \tag{5.6}$$

$$\gamma(\mathbf{c}_1 \to t_1, \ldots, \mathbf{c}_m \to t_m, * \to t) := ite(\gamma(\mathbf{c}_1), t_1, \ldots ite(\gamma(\mathbf{c}_m), t_m, t) \ldots) \tag{5.7}$$

In the remainder of this section, we assume we are given a set of ground clauses F, a satisfying assignment M for F, a congruence closure E^* for M and an evaluation map $\mathcal{A}_M$ for M. We will write $[\![t]\!]$ to denote the evaluation of a term t that is composed of values from $\mathbf{V}$, the equality predicate $\approx$ and logical connectives such as *ite*, and $\wedge$. For all such terms, we assume this evaluation is performed in the obvious way.

Example 9 *Say we wish to define a unary predicate P that is true for v but false otherwise. Predicate P can be represented by the definition $(v) \to \mathbf{true}$, $(*) \to \mathbf{false}$.*

Example 10 *Say we wish to define a binary function f that maps every pair whose first argument is v to the value 1, every pair whose second argument is w to 2 and the remaining pairs to 0. Function f can be represented by the definition $D_f := (v, *) \to 1$, $(*, w) \to 2$, $(*, *) \to 0$. To interpret the term $f(v, w)$, notice that $\gamma(D_f)[(v, w)]$ is the term $ite(v \approx v, 1, ite(w \approx w, 2, 0))$, and $[\![ite(v \approx v, 1, ite(w \approx w, 2, 0))]\!] = 1$.*

Notice that the interpretation of a definition may depend on the order in which

entries occur. In the previous example, if our definition D_f was $(*, w) \to 2$, $(v, *) \to 1$, $(*, *) \to 0$, then $f(v, w)$ would evaluate to 2 instead.

Definition 6 (Σ-map) *For a signature Σ, a* Σ-map *is a mapping $\mathbf{D}_\Sigma$ from all uninterpreted functions $f \in \Sigma$ with range sort S to a corresponding complete definition D_f of the same arity, whose range is $\mathbf{V}^S$. Given a Σ-map $\mathbf{D}_\Sigma$, we can define a Σ structure $\mathcal{M}$, where for each uninterpreted function symbol $f \in \Sigma$:*

$$f^{\mathcal{M}} = \lambda \mathbf{x}.[\![\gamma(D_f)[\mathbf{x}]]\!]$$

In this case, we will say that $\mathcal{M}$ is induced by the Σ-map $\mathbf{D}_\Sigma$.

5.3.3 Constructing Function Definitions

In our approach, we only consider models that are induced by Σ-maps. As shown in the following, every normal model (see Section 4.1) is induced by a Σ-map, and thus this restriction is not limiting. We show how function definitions D_f are constructed for each function f in our signature Σ.

As mentioned, a satisfying assignment M contains asserted ground equalities, for which we construct an evaluation map $\mathcal{A}_M$ containing model assignments from terms in M to values from $\mathbf{V}$. For each entry of the form $f(t_1, \ldots, t_n) \mapsto v$ in $\mathcal{A}_M$, we will associate a corresponding entry of the form $\mathbf{c} \to v$ in D_f, where $\mathbf{c}$ is an n-tuple. For the purposes of choosing default values, we associate to each sort S a distinguished ground Σ-term e^S, which we will write ambiguously here just as e when convenient. Given an n-tuple of values $\mathbf{c}$, we write $\mathbf{c}^\forall$ to denote the n-tuple of abstract

values such that for all $i = 1, \ldots, n$:

$$\mathbf{c}^{\forall}.i = * \qquad \text{if } \mathbf{c}.i = \mathcal{A}_M(e) \tag{5.8}$$

$$\mathbf{c}^{\forall}.i = \mathbf{c}.i \qquad \text{otherwise} \tag{5.9}$$

In other words, $\mathbf{c}^{\forall}$ replaces all occurrences of the value of our distinguished term with the abstract value $*$.

We now provide a method for constructing function definitions D_f for each $f \in \Sigma$, which we will use to represent the interpretation of f in candidate model $\mathcal{M}$.

Construction of D_f: Let U be a subset of the ground terms $\mathbf{T}_M$. Then, D_f is a definition containing a minimal number of entries that satisfies the following set of constraints.

$$\begin{aligned}
(\mathcal{A}_M(t_1), \ldots, \mathcal{A}_M(t_n))^{\forall} \rightarrow \mathcal{A}_M(f(t_1, \ldots, t_n)) &\in D_f, \quad \forall f(t_1, \ldots, t_n) \in U \\
(\mathcal{A}_M(t_1), \ldots, \mathcal{A}_M(t_n)) \rightarrow \mathcal{A}_M(f(t_1, \ldots, t_n)) &\in D_f, \quad \forall f(t_1, \ldots, t_n) \in \mathbf{T}_M \setminus U \\
(*, \ldots, *) \rightarrow v &\in D_f, \quad \text{for some } v
\end{aligned}$$

To show our construction of D_f is well defined, say our construction gives us the constraints $\mathbf{c_1} \rightarrow v_1 \in D_f$, …, $\mathbf{c_m} \rightarrow v_m \in D_f$. We sort these entries $\mathbf{c_{i1}} \rightarrow v_{i1}, \ldots, \mathbf{c_{im}} \rightarrow v_{im}$ such that $\mathbf{c_{ij}} \not\succeq \mathbf{c_{ik}}$ for $j < k$. Sorting entries in this way is always possible, since $\succeq$ is a partial order, and because $\mathbf{c} \rightarrow v_i$ and $\mathbf{c} \rightarrow v_j$ are not both in D_f where $v_i \neq v_j$. We show the latter by contradiction. If $\mathbf{c} \rightarrow v_i$ and $\mathbf{c} \rightarrow v_j$ are both in D_f, then we have that for two terms, $\mathcal{A}_M(f(t_1, \ldots, t_n)) = v_i$ and

$\mathcal{A}_M(f(s_1,\ldots,s_n)) = v_j$. Since M is a satisfying assignment and $v_i \neq v_j$, then it must be that $\mathcal{A}_M(t_i) \neq \mathcal{A}_M(s_i)$ for some i. This is a contradiction: if $\mathbf{c}.i$ is $*$, we have that $\mathcal{A}_M(t_i) = \mathcal{A}_M(s_i) = \mathcal{A}_M(e)$; otherwise, we have $\mathcal{A}_M(t_i) = \mathcal{A}_M(s_i) = \mathbf{c}.i$. Thus, our construction of D_f is well defined. Moreover, it is consistent with $\mathcal{A}_M$ as shown in the following lemma.

Lemma 4 *For all terms of the form $f(t_1,\ldots,t_n)$ in $\mathbf{T}_M$, $[\![\gamma(D_f)[\mathcal{A}_M(t_1),\ldots,\mathcal{A}_M(t_n)]]\!]$ $= \mathcal{A}_M(f(t_1,\ldots,t_n))$.*

Proof: Since $f(t_1,\ldots,t_n)$ in $\mathbf{T}_M$, we have that $\mathbf{t} \to \mathcal{A}_M(t)$ is in D_f for some $\mathbf{t} \succeq (\mathcal{A}_M(t_1),\ldots,\mathcal{A}_M(t_n))$. By contradiction assume that $[\![\gamma(D_f)[\mathcal{A}_M(t_1),\ldots,\mathcal{A}_M(t_n)]]\!]$ $= w \neq \mathcal{A}_M(t)$. Let $\mathbf{s} \to w$ be the first entry in D_f whose condition is compatible with $(\mathcal{A}_M(t_1),\ldots,\mathcal{A}_M(t_n))$, and let $s = f(s_1,\ldots,s_n)$ be the term associated with this entry, where $\mathcal{A}_M(s) = w$, and $\mathbf{s} \succeq (\mathcal{A}_M(s_1),\ldots,\mathcal{A}_M(s_n))$.

Since $\mathcal{A}_M$ is an evaluation map, we have that $\mathcal{A}_M(s_i) \neq \mathcal{A}_M(t_i)$ for some i. Since $\mathcal{A}_M(t_i)$ is compatible with $\mathbf{s}.i$ and $\mathbf{s}.i \succeq \mathcal{A}_M(s_i)$, we have that $\mathbf{s}.i$ must be $*$, and thus $s \in U$. Additionally, we have that $\mathbf{s}.j \not\succeq \mathbf{t}.j$ for some j, since $\mathbf{s} \to \mathcal{A}_M(s)$ comes before $\mathbf{t} \to \mathcal{A}_M(t)$ in D_f. Since $\mathcal{A}_M(t_j)$ is compatible with both $\mathbf{s}.j$ and $\mathbf{t}.j$, we have that $\mathbf{s}.j$ is $\mathcal{A}_M(t_j)$ and $\mathbf{t}.j$ is $*$. Since t_j must be e^S, we have that $s \notin U$, a contradiction. ■

Theorem 4 *Let $\mathbf{D}_\Sigma$ be the Σ-map mapping function symbols f to definitions D_f, as constructed in this section. The Σ-structure $\mathcal{M}$ induced by $\mathbf{D}_\Sigma$ satisfies F.*

Proof: Let E^* be the congruence closure for M from which $\mathcal{A}_M$ is constructed, and

let R be the union of the equalities in E^* and the disequalities $\{t \not\approx s \mid t, s \in \mathbf{T}_M, t \approx s \notin E^*\}$. For all terms t in $\mathbf{T}_M$, from Lemma 4, and by induction on the structure of t, we have that $\mathcal{M}[\![t]\!] = \mathcal{A}_M(t)$. Since $\mathcal{A}_M$ was constructed from E^*, we have that $\mathcal{M} \models R$, and furthermore $R \models_T M$ and $M \models F$. ■

Notice that our model construction is parameterized by our choice of the set U, which we call our *selected terms*. Two obvious choices for U are to choose no terms, and to choose all terms in $\mathbf{T}_M$. The resultant models from these choices we will call *simple* and *fragmented* models respectively. The latter we call fragmented since they are induced by function definitions D_f that map pieces of domain of f to values based on each equality in M, instead of just mapping points to values.

Example 11 *Say our satisfying assignment is*

$$M = \{g(b) \approx a,\ h(a) \approx b,\ h(b) \approx b,\ a \approx f(a), f(a) \not\approx g(a)\}$$

where all terms have the same sort and a is our distinguished term for that sort. Say our congruence closure E^* for M consists of the equivalence classes $\{a, g(b), f(a)\}$, $\{b, h(a), h(b)\}$, and $\{g(a)\}$. We construct a candidate model $\mathcal{M}$ from M that is induced by a Σ-map. Assuming the values $\mathbf{V}$ of $\mathcal{M}$ are $\{a,\ b,\ g(a)\}$, and $U = \mathbf{T}_M$, our definitions for f, g, and h would be:

$$\begin{aligned} D_f &= (*) \to a, \\ D_g &= (b) \to a, (*) \to g(a), \\ D_h &= (b) \to b, (*) \to b. \end{aligned}$$

■

5.3.4 Simplifying Function Definitions

The efficiency of our approach when checking models induced by Σ-maps will be dependent upon the size of the representation of our model. Thus, it will be crucial to simplify definitions D_f in Σ-maps. We do this as follows.

First, an entry $\mathbf{c} \to t$ is *redundant* in definition D if :

- $D := \ldots, \mathbf{c} \to t, \mathbf{d_1} \to s_1, \ldots, \mathbf{d_n} \to s_n, \mathbf{e} \to t, \ldots,$
- for each $1 \leq i \leq n$, either $\mathbf{c}$ is not compatible with $\mathbf{d_i}$, or $t = s_i$, and
- $\mathbf{e} \succeq \mathbf{c}$

When an entry $\mathbf{c} \to t$ is redundant in definition D, notice that removing the entry $\mathbf{c} \to t$ from D does not affect the meaning of D.

Example 12 *Say D is the definition $(v, *) \to 1, (w, w) \to 0, (*, *) \to 1$. The entry $(v, *) \to 1$ is redundant in D.*

To simplify a definition D, we wish to remove all redundant entries from it. While constructing a definition, we maintain a status flag for each of its entries, either yes, no, or unknown. When an entry $\mathbf{e} \to t$ is appended to D, we mark the entry's status as unknown. Then, for each $\mathbf{c} \to s$ in D whose status is unknown and $\mathbf{c}$ is compatible with $\mathbf{e}$, if $s = t$ and $\mathbf{e} \succeq \mathbf{c}$, we mark $\mathbf{c} \to s$ as yes. Otherwise, if $s \neq t$, we mark $\mathbf{c} \to s$ as no. When we are finished constructing the definition, we remove all entries marked yes.

Unfortunately the above procedure does not recognize certain cases when an entry is unreachable, that is, when no ground tuple of values witnesses that entry. For instance, consider the finite domain with two elements $\{v_1, v_2\}$, and the definition $(v_1) \rightarrow \textbf{true}, (v_2) \rightarrow \textbf{true}, * \rightarrow \textbf{false}$. Clearly, the entry $* \rightarrow \textbf{false}$ is unreachable. Our implementation uses various (incomplete) heuristics for recognizing and eliminating some cases of this form.

5.4 Model-Based Quantifier Instantiation

We have now seen how satisfying assignments M can be found for clause sets in DPLL$(T_1, \ldots, T_m)$, and how candidate models are constructed from M. In this section, we describe how candidate models are used while performing quantifier instantiation heuristics.

Recall that for a candidate model $\mathcal{M}$, our quantifier instantiation heuristic will choose a set of substitutions $I_{\mathbf{x}}$ to terms taken from $\mathbf{V}$ (the domain elements of $\mathcal{M}$) for each active quantified formula Q in M. When choosing the set $I_{\mathbf{x}}$, a naive approach is to chose all combinations of the properly sorted domain elements from $\mathbf{V}$. Doing so requires k^n instantiations for a quantifier over n variables each ranging over a domain of size k, which is feasible only if both k and n are small. An improved and significantly more scalable approach can be used if we can recognize when sets of ground instances are already satisfied by the current candidate model and hence can be ignored. This approach is known as *model-based quantifier instantiation*.

A previous approach for model-based quantifier instantiation, as implemented in the SMT solver Z3, uses the SMT solver itself as an oracle. That is, a separate copy

of the SMT solver is run on another query to determine whether a candidate model $\mathcal{M}$ satisfies each quantified formula. If it does not, it adds a single instance that is falsified by $\mathcal{M}$ to the current clause set. While simple to implement, this approach incurs performance overhead for both constructing the corresponding query as well as the initialization of the oracle. Our approach for model-based instantiation instead relies upon specialized data structures when checking candidate models and choosing instantiations, and may add more than one instantiation per invocation.

We present two new algorithms for model-based quantifier instantiation, both of which can be used in the context of finite model finding. For both, we describe which substitutions $I_{\mathbf{x}}$ are chosen for quantified formulas from Q, given a candidate model $\mathcal{M}$. As we will show, if this set is empty, then $\mathcal{M}$ satisfies all instances of quantified formulas from Q, and thus all formulas in Q.

5.4.1 Algorithm for Generalizing Evaluations

We describe a model-based quantifier instantiation method in this section that identifies entire sets of instances as satisfiable in $\mathcal{M}$ without actually generating and checking those instances individually [54]. The main idea is to determine the satisfiability in $\mathcal{M}$ of some ground instance $\varphi\sigma$ of a quantified formula $\forall \mathbf{x}\, \varphi \in Q$, generalize $\varphi\sigma$ to a set of J of instances equisatisfiable with $\varphi\sigma$ in $\mathcal{M}$, and then look for further instances only outside that set. The set J is computed by identifying which variables of φ actually matter in determining the satisfiability of $\varphi\sigma$. Technically, for each $\psi = \forall \mathbf{x}\, \varphi \in Q$, substitution $\sigma = \{\mathbf{x} \mapsto \mathbf{v}\}$ into $\mathbf{V}$, and ground instance $\varphi' = \varphi\sigma$ of ψ, if $\mathcal{M} \models \varphi'$ we compute a partition of $\mathbf{x}$ into $\mathbf{x}_1$ and $\mathbf{x}_2$ and a corresponding

```
proc eval(M, t, σ) ≡
  match t with
    | f(t_1, ..., t_n) →  for j = 1, ..., n
                             let (v_j, X_j) = eval(M, t_j, σ)
                          end
                          choose a critical argument subset C of {1, ..., n}
                          return (f^M(v_1, ..., v_n), ⋃_{i∈C} X_i)
    | x →  return (σ(x), {x})
```

Figure 5.2. The eval procedure for candidate model $\mathcal{M}$.

partition of $\mathbf{v}$ into $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $\mathcal{M} \models \forall \mathbf{x}_2\, \varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$; similarly, if $\mathcal{M} \not\models \neg\varphi'$ we compute a partition such that $\mathcal{M} \not\models \forall \mathbf{x}_2\, \neg\varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$. In either case, we then know that all ground instances of $\varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$ over $\mathbf{V}$ are equisatisfiable with φ' in $\mathcal{M}$, and so it is enough to consider just φ' in lieu of all them. We will refer to the elements of $\mathbf{x}_1$ above as a set of *critical variables for* φ *(under* σ*)*—although strictly speaking this is a misnomer as we do not insist that $\mathbf{x}_1$ be minimal.

5.4.1.1 Generalizing Evaluations

Treating quantifier-free formulas as Boolean terms (which evaluate to either **true** or **false** in a Σ-structure depending on whether they are satisfied by the model or not), we developed a general procedure that, given the Σ-map of a candidate model $\mathcal{M}$, a term t, and a substitution σ over t's variables, computes and returns both the value of $t\sigma$ in $\mathcal{M}$ and a set of critical variables for σ.

The procedure, defined recursively over the input term and assuming a prefix form for the logical operators as well, is sketched in Figure 5.2. When evaluating a non-variable term $f(t_1, \ldots, t_n)$, eval determines a *critical argument subset* C for it.

This is a subset of $\{1, \ldots, n\}$ such that the term $f(s_1, \ldots, s_n)$ denotes a constant function in $\mathcal{M}$ where each s_i is the value computed by eval for t_i if $i \in C$, and is a unique variable otherwise. If f is a logical symbol, the choice of C is dictated by the symbol's semantics. For instance, for $\approx(t_1, t_2)$, C is $\{1, 2\}$; for $\vee(t_1, \ldots, t_n)$, it is $\{1, \ldots, n\}$ if the disjunction evaluates to **false**; otherwise, we may choose $\{i\}$ for some i where t_i evaluates to **true**. If f is a function symbol of Σ, eval computes C by first constructing a custom index data structure for interpreting applications of f to values. The key feature of this data structure is that it uses information on the sets $X_1, \ldots X_n$ to choose an evaluation order for the arguments of f. For example, given the term $t = f(g(x, y, z), v_2, h(x))$, say that eval computes the values v_1, v_2, v_3 and the critical variable sets $\{x, y, z\}$, $\emptyset$, $\{x\}$ for the three arguments of f, respectively. With those sets, it will use the evaluation order $(2, 3, 1)$ for those arguments—meaning that the second argument is evaluated first, then the third, etc. Using the index data structure, it will first determine if $f(x_1, v_2, x_3)$ has a constant interpretation in $\mathcal{M}$. If so, then the evaluation of t depends on none of its variables, and the returned set of critical variables for t will be $\emptyset$. Otherwise, if $f(x_1, v_2, v_3)$ has a constant interpretation in M, then the evaluation of t depends on $\{x\}$, or else it depends on the entire variable set $\{x, y, z\}$.

The next example gives more details on the whole process of generalizing a ground instance to a set of ground instances equisatisfiable with it in the given model.

Example 13 *Let* $Q = \{\forall x_1\, x_2.\, f(x_2) \approx g(x_1, b) \vee h(x_1, x_2) \not\approx b\}$. *Consider a candidate model* $\mathcal{M}$ *induced by a* Σ*-map containing the following definitions :*

$$D_g = (a,a) \to c,\ (*,b) \to a,\ (*,*) \to b$$
$$D_f = (b) \to b,\ (*) \to a$$
$$D_h = (*,*) \to b$$

The table below shows the bottom-up calculation performed by eval *on the formula* $\varphi = f(x_2) \approx g(x_1,b) \vee h(x_1,x_2) \not\approx b$ *with* $\mathcal{M}$ *above and* $\sigma = \{x_1 \mapsto a, x_2 \mapsto a\}$*.*

input	**output**	**critical arg. subset**
x_2	$(a, \{x_2\})$	
x_1	$(a, \{x_1\})$	
b	$(b, \emptyset)$	$\emptyset$
$f(x_2)$	$(a, \{x_2\})$	$\{1\}$
$g(x_1, b)$	$(a, \emptyset)$	$\{2\}$
$h(x_1, x_2)$	$(b, \emptyset)$	$\emptyset$
$f(x_2) \approx g(x_1, b)$	$(\mathbf{true}, \{x_2\})$	$\{1, 2\}$
$h(x_1, x_2) \not\approx b$	$(\mathbf{false}, \emptyset)$	$\{1, 2\}$
$f(x_2) \approx g(x_1, b) \vee h(y, x_2) \not\approx b$	$(\mathbf{true}, \{x_2\})$	$\{1\}$

For most entries in the table the evaluation is straightforward. For a more interesting case, consider the evaluation of $g(x_1, b)$*. First, the arguments of* g *are evaluated, respectively to* $(a, \{x_1\})$ *and* $(b, \emptyset)$*, but with evaluation order* $(2, 1)$*. Using an indexing data structure built from* D_g *for the evaluation order* $(2, 1)$*, we determine that* $g(x, b)$ *has constant value* a *for all* x*. Hence we return an empty set of critical variables for* $g(x_1, b)$*.*

Similarly, the fact that eval *returns* $(\mathbf{true}, \{x_2\})$ *for the original input formula* φ *and the substitution* $\sigma = \{x_1 \mapsto a,\ x_2 \mapsto a\}$ *means that we were able to determine that all ground instances of* $\varphi\{x_2 \mapsto a\} = (f(a) \approx g(x_1, b) \vee h(x_1, a) \not\approx b)$*, not just the instance* $\varphi\sigma$*, are satisfied in* $\mathcal{M}$*. We can then use this information to completely avoid generating and checking those instances.* ■

5.4.1.2 Choosing Instantiations

For any given quantified formula ψ, the eval procedure allows us to identify a set of instances over $\mathbf{V}$ that can be represented by a single one, as far as satisfiability in the candidate model $\mathcal{M}$ is concerned. The next question then is how to generate a set I of instances that together represent *all* instances of ψ over $\mathbf{V}$ that are falsified by $\mathcal{M}$. This kind of exhaustiveness is crucial because it allows us to conclude that $\mathcal{M} \models \psi$ by just checking that I is empty.

We present a procedure that relies on eval for computing the set I above, or rather, a set of substitutions for generating the elements of I from ψ. The procedure is fairly unsophisticated and quite conservative in its choice of representative instances, which makes it very simple to implement and prove correct. Its main shortcoming is that it does not take full advantage of the information provided by eval, and so may end up producing more representative instances than needed in many cases.

Let $\psi = \forall \mathbf{x}\, \varphi \in Q$ with $\mathbf{x} = (x_1, \ldots, x_n)$. For $i = 1, \ldots, n$, let S_i be the sort of x_i and let $\mathbf{V}_\mathbf{x} = \mathbf{V}_{S_1} \times \cdots \times \mathbf{V}_{S_n}$. For each $S \in \{S_1, \ldots, S_n\}$, let $<_S$ be an arbitrary total ordering over the values $\mathbf{V}_S$ of sort S. Let $<$ be the *lexicographic* extension of these orderings to the tuples in $\mathbf{V}_\mathbf{x}$ and observe that $\mathbf{V}_\mathbf{x}$ is totally ordered by $<$. We write $\mathbf{v}_{min}$ to denote the minimum of $\mathbf{V}_\mathbf{x}$ with respect to this ordering.

For every $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbf{V}_\mathbf{x}$ Let $\mathsf{next}_i(\mathbf{v})$ denote the smallest tuple $\mathbf{u}$ with respect to $<$ such that $\mathbf{v}.j <_{S_j} \mathbf{u}.j$ for some $1 \leq j \leq n + 1 - i$, if such a tuple exists, and denote $\mathbf{v}_{min}$ otherwise (including when $i > n$). For instance, with $n = 3$, $S_1 = S_2 = S_3$ and $\mathbf{V}_{S_1} = \{a, b\}$ with $a <_{S_1} b$, we have that $\mathsf{next}_1(a, a, a) = (a, a, b)$,

```
proc choose_instances(M, φ, x) ≡
  I_x := ∅;  t := v_min
  do
    (v, {x_{i_1}, ..., x_{i_m}}) := eval(M, φ, {x ↦ t})
    if v = false then  I_x := I_x ∪ {{x ↦ t}}
    end
    t := next_i(t) where i is the minimum of {i_1, ..., i_m, n + 1}
  while t ≠ v_min
  return I_x
```

Figure 5.3. The $\mathsf{choose_instances}$ procedure. We assume $\mathbf{x} = (x_1, \ldots, x_n)$.

$\mathsf{next}_2(a, a, a) = (a, b, a)$, $\mathsf{next}_2(a, b, a) = (b, a, a)$, and $\mathsf{next}_2(b, b, a) = \mathbf{v}_{min} = (a, a, a)$. Note that except in the case that $\mathsf{next}_i(\mathbf{v})$ is $\mathbf{v}_{min}$, we have that $\mathbf{v} < \mathsf{next}_i(\mathbf{v})$.

Our instantiation heuristic $\mathcal{H}$ chooses substitutions $I_{\mathbf{x}}$ based on the procedure $\mathsf{choose_instances}$ described in Figure 5.3, which takes in a quantifier-free formula φ with variables $\mathbf{x}$ and returns a set $I_{\mathbf{x}}$ of substitutions σ for $\mathbf{x}$ such that $\mathcal{M} \not\models \varphi\sigma$. At each execution of its loop the procedure implicitly determines with eval a set of I of instances of φ that are equisatisfiable with $\varphi\{\mathbf{x} \mapsto \mathbf{v}\}$ in $\mathcal{M}$, where $\mathbf{v}$ is the tuple stored in the program variable t. The next value t_{next} for t is a greater tuple chosen to maintain the invariant that all the tuples between t and t_{next} generate instances of φ that are in I. To see that, it suffices to observe that these tuples differ from $\mathbf{t}$ only in positions that correspond to non-critical variables of φ, namely those before position i where x_i is the first critical variable of φ in the enumeration $x_1, \ldots, x_n$. This observation is the main argument in the proof of the following result.

Lemma 5 *Let $\mathbf{v}_0, \ldots, \mathbf{v}_m$ be all values successively taken by the variable $\mathbf{t}$ at the beginning of the loop in* $\mathsf{choose_instances}$*. Let v_{max} be the maximum element of $\mathbf{V}_{\mathbf{x}}$.*

Then for all $i = 1, \ldots, m$,

1. $\mathbf{v}_{i-1} < \mathbf{v}_i$,

2. *for all* $\mathbf{u}$ *with* $\mathbf{v}_{i-1} \leq \mathbf{u} < \mathbf{v}_i$, $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ *iff* $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_{i-1}\}$,

3. *for all* $\mathbf{u}$ *with* $\mathbf{v}_m \leq \mathbf{u} \leq \mathbf{v}_{max}$, $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ *iff* $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_m\}$.

Proof: (Sketch) The first statement is immediate since for all $i = 1 \ldots m$, we have $v_i = \mathsf{next}_k(\mathbf{v}_{i-1})$ for some k and $v_i \neq \mathbf{v}_{min}$. To show the second statement for an i, assume $\mathbf{v}_i = \mathsf{next}_k(\mathbf{v}_{i-1})$ for some k. For each $\mathbf{u}$ where $\mathbf{v}_{i-1} \leq \mathbf{u} < \mathbf{v}_i$, we have that $\mathbf{u}.j = \mathbf{v}_{i-1}.j$ for all $j \geq k$. For all $j < k$, the eval procedure determined that the variable x_j was not a critical variable for φ. Since $\mathbf{u}$ and $\mathbf{v}_{i-1}$ vary on only these variables, we have $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ iff $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_{i-1}\}$. The third statement holds for similar reasons as the second. ■

Proposition 2 *The set* $I_{\mathbf{x}}$ *returned by* choose_instances$(\mathcal{M}, \varphi, \mathbf{x})$ *is empty if and only if* $\mathcal{M} \models \forall \mathbf{x}\, \varphi$.

Proof: Due to the previous lemma, when there exists an instance of φ that is falsified by $\mathcal{M}$, then choose_instances will consider at least one $\mathbf{v}_i$ for which $\varphi\{\mathbf{x} \mapsto \mathbf{v}_i\}$ evaluates to **false**, and hence it will return at least one instance. Conversely, if all instances of φ are satisfied by $\mathcal{M}$, then all instances of φ considered by choose_instances evaluate to **true**, and hence it will return no instances. ■

We remark that, for our model finding purposes, there is no need for the procedure choose_instances to compute the full set $I_{\mathbf{x}}$ once it contains at least one

substitution. Any non-empty subset would suffice to trigger a (more incremental) revision of the current candidate model $\mathcal{M}$. That said, our current implementation does compute the whole set and adds all the corresponding instances to Q before computing another model for it. Our experiments show that computing and using one substitution at a time is worse for overall performance than computing and using the full set $I_{\mathbf{x}}$.

5.4.2 Algorithm for Computing Interpretations for Terms

In this section, we present an alternative method for choosing instantiations, which will be based on constructing data structures that represent the interpretation of (non-ground) terms in $\mathcal{M}$. We extend our representation of function definitions from Section 5.3.2 to non-ground terms t possibly containing variables $\mathbf{x} = (x_1, \ldots, x_n)$, written $D_{\lambda \mathbf{x}.t}$, which we use to compute the interpretation of ground instances of t in $\mathcal{M}$. We will call $D_{\lambda \mathbf{x}.t}$ a *term definition*. Due to our construction, it will be the case that $[\![\gamma(D_{\lambda \mathbf{x}.t})[\mathbf{v}]]\!]$ is equal to $\mathcal{M}\{\mathbf{x} \mapsto \mathbf{v}\}[\![t]\!]$ for all n-tuples of ground values $\mathbf{v}$. To ensure $\gamma(D_{\lambda \mathbf{x}.t})[\mathbf{v}]$ is well-sorted, we ensure all entries $\mathbf{c} \rightarrow w$ in $D_{\lambda \mathbf{x}.t}$ are such that $\mathbf{c}.i$ has the sort of x_i for $i = 1, \ldots, n$. First, we define several operations over function and term definitions.

5.4.2.1 Operations on Definitions

This section explains various operations over definitions used for computing the interpretation of terms in a candidate model. Throughout the remainder of the section, we extend definitions from Section 5.3.2 to contain entries of the form $\mathbf{c} \rightarrow \mathbf{t}$, where $\mathbf{t}$ is a tuple of terms. For the purposes of interpreting definitions of this form,

we extend our evaluation function such that for a tuple $\mathbf{t} = (t_1, \ldots, t_n)$, we have $[\![\mathbf{t}]\!] = ([\![t_1]\!], \ldots, [\![t_n]\!])$.

Definition 7 (Entry Append) *The append operation for definitions and entries, written* $D \cdot \mathbf{c} \to t$*, is defined as follows:*

$$\begin{aligned} D \cdot \mathbf{c} \to t &:= D && \textit{if } \mathbf{c} \textit{ is } \bot, \textit{ or } \mathbf{d} \succeq \mathbf{c} \textit{ for some } \mathbf{d} \to s \in D \\ D \cdot \mathbf{c} \to t &:= D, \mathbf{c} \to t && \textit{otherwise} \end{aligned}$$

Notice that for all definitions D, we have that $D \cdot \mathbf{c} \to t$ is also a definition.

Definition 8 (Product) *For definitions* $D_{\lambda\mathbf{x}.t_1} = \mathbf{c_1} \to t_1, \ldots, \mathbf{c_n} \to t_n$ *and* $D_{\lambda\mathbf{x}.t_2} = \mathbf{d_1} \to s_1, \ldots, \mathbf{d_m} \to s_m$*, we define their Cartesian product, written* $D_{\lambda\mathbf{x}.(t_1,t_2)}$*, as a definition of arity* m *to pairs, as follows:*

$$\emptyset \cdot \mathbf{c_1} \triangle \mathbf{d_1} \to (t_1, s_1) \ \cdot \ldots \ \cdot \mathbf{c_1} \triangle \mathbf{d_m} \to (t_1, s_m) \ \cdot$$

$$\ldots$$

$$\mathbf{c_n} \triangle \mathbf{d_1} \to (t_n, s_1) \ \cdot \ldots \ \cdot \mathbf{c_n} \triangle \mathbf{d_m} \to (t_n, s_m)$$

Example 14 *Say* $D_{\lambda x_1.t_1}$ *is* $(v) \to 0$, $(*) \to 1$ *and* $D_{\lambda x_1.t_2}$ *is* $(w) \to 0$, $(*) \to 1$. *Then,* $D_{\lambda x_1.(t_1,t_2)}$ *is the definition* $(v) \to (0,1)$, $(w) \to (1,0)$, $(*) \to (1,1)$.

Example 15 *Say* $D_{\lambda x_1\, x_2.t_1}$ *is* $(v,*) \to 0$, $(*,*) \to 1$ *and* $D_{\lambda x_1\, x_2.t_2}$ *is* $(v,w) \to 0$,$(w,*) \to 1$,$(*,*) \to 2$. *Then,* $D_{\lambda x_1\, x_2.(t_1,t_2)}$ *is the definition* $(v,w) \to (0,0)$, $(v,*) \to (0,2)$, $(w,*) \to (1,1)$, $(*,*) \to (1,2)$.

Lemma 6 *If* $D_{\lambda\mathbf{x}.t_1}$ *and* $D_{\lambda\mathbf{x}.t_2}$ *are complete definitions, then* (*i*) $D_{\lambda\mathbf{x}.(t_1,t_2)}$ *is a complete definition, and* (*ii*) $[\![\gamma(D_{\lambda\mathbf{x}.(t_1,t_2)})[\mathbf{v}]]\!] = ([\![\gamma(D_{\lambda\mathbf{x}.t_1})[\mathbf{v}]]\!], [\![\gamma(D_{\lambda\mathbf{x}.t_2})[\mathbf{v}]]\!])$*, for all values* $\mathbf{v}$.

Proof: Let $D_{\lambda\mathbf{x}.t_1}$ be $\mathbf{c_1} \to s_1, \ldots, \mathbf{c_n} \to s_n$ and $D_{\lambda\mathbf{x}.t_2}$ be $\mathbf{d_1} \to r_1, \ldots, \mathbf{d_m} \to r_m$. To show (i), since $D_{\lambda\mathbf{x}.t_1}$ and $D_{\lambda\mathbf{x}.t_2}$ are complete, then $\mathbf{c_n}$ and $\mathbf{d_m}$ are $*$ and thus $D_{\lambda\mathbf{x}.(t_1,t_2)}$ contains the entry $* \to (s_n, r_m)$.

To show (ii), since $D_{\lambda\mathbf{x}.t_1}$ is complete, say $[\![\gamma(\mathbf{c_j})[\mathbf{v}]]\!]$ is true and $[\![\gamma(\mathbf{c_1})[\mathbf{v}]]\!] \ldots [\![\gamma(\mathbf{c_{j-1}})[\mathbf{v}]]\!]$ are false. Similarly, say $[\![\gamma(\mathbf{d_k})[\mathbf{v}]]\!]$ is true, and $[\![\gamma(\mathbf{d_1})[\mathbf{v}]]\!] \ldots [\![\gamma(\mathbf{d_{k-1}})[\mathbf{v}]]\!]$ are false. We know that $[\![\gamma(\mathbf{c_i} \bigtriangleup \mathbf{d})[\mathbf{v}]]\!] = [\![\gamma(\mathbf{c_i})[\mathbf{v}]]\!] \wedge [\![\gamma(\mathbf{d})[\mathbf{v}]]\!]$ is false for any $\mathbf{d}$ where $1 \le i < j$. Similarly $[\![\gamma(\mathbf{c} \bigtriangleup \mathbf{d_i})[\mathbf{v}]]\!]$ is false for any $\mathbf{c}$ where $1 \le i < k$. Since $[\![\gamma(\mathbf{c_j} \bigtriangleup \mathbf{d_k})[\mathbf{v}]]\!] = [\![\gamma(\mathbf{c_j})[\mathbf{v}]]\!] \wedge [\![\gamma(\mathbf{d_k})[\mathbf{v}]]\!]$ is true, $[\![\gamma(D_{\lambda\mathbf{x}.(t_1,t_2)})[\mathbf{v}]]\!] = (s_j, r_k) = ([\![\gamma(D_{\lambda\mathbf{x}.t_1})[\mathbf{v}]]\!], [\![\gamma(D_{\lambda\mathbf{x}.t_2})[\mathbf{v}]]\!])$. ■

The n-fold Cartesian product of definitions $D_{\lambda\mathbf{x}.t_1}, \ldots D_{\lambda\mathbf{x}.t_n}$, written $D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)}$, is defined inductively for $n \ge 0$:

$$D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)} := * \to () \qquad \text{if } n = 0 \tag{5.10}$$

$$D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)} := D_{\lambda\mathbf{x}.((t_1,\ldots,t_{n-1}),t_n)} \qquad \text{otherwise} \tag{5.11}$$

We will treat $D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)}$ as a definition whose values are n-tuples, that is, we flatten the left-associative chain of pairs occurring in the range of $D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)}$.

Example 16 *Say* $D_{\lambda x_1.t_1}$ *is* $(v) \to v, (*) \to w$, $D_{\lambda x_1.t_2}$ *is* $(w) \to w, (*) \to v$, *and* $D_{\lambda x_1.t_3}$ *is* $(*) \to w$. *Then,* $D_{\lambda x_1.()} = (*) \to ()$, $D_{\lambda x_1.((),t_1)} = D_{\lambda x_1.(t_1)} = (v) \to (v), (*) \to (w)$, $D_{\lambda x_1.((t_1),t_2)} = D_{\lambda x_1.(t_1,t_2)} = (v) \to (v,v), (w) \to (w,w), (*) \to (w,v)$, *and* $D_{\lambda x_1.((t_1,t_2),t_3)} = D_{\lambda x_1.(t_1,t_2,t_3)} = (v) \to (v,v,w), (w) \to (w,w,w), (*) \to (w,v,w)$.

```
proc compose(c → (t_1, ..., t_n), (d_1, ..., d_n) → v) ≡
  if n = 0
    return c → v
  else if t_n is x_j, and c.j is compatible with d_n
    return compose(c △_j d_n → (t_1, ..., t_{n−1}), (d_1, ..., d_{n−1}) → v)
  else if t_n is v, and v is compatible with d_n
    return compose(c → (t_1, ..., t_{n−1}), (d_1, ..., d_{n−1}) → v)
  else
    return ⊥ → v
  end
```

Figure 5.4. Method for computing the composition for entries. Term t_i is either a value or a variable from $\mathbf{x} = (x_1, \ldots x_m)$ where d_i has the sort of t_i for each $i = 1, \ldots n$, and $\mathbf{c}$ is an m-tuple where $\mathbf{c}.j$ has the sort of x_j for each $j = 1, \ldots, m$.

Lemma 7 *If $D_{\lambda \mathbf{x}.t_1} \ldots D_{\lambda \mathbf{x}.t_n}$ are complete definitions, then (i) $D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)}$ is a complete definition, and (ii) $[\![\gamma(D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)})[\mathbf{v}]]\!] = ([\![\gamma(D_{\lambda \mathbf{x}.t_1})[\mathbf{v}]]\!], \ldots, [\![\gamma(D_{\lambda \mathbf{x}.t_n})[\mathbf{v}]]\!])$, for all $\mathbf{v}$.*

Proof: By induction on n using Lemma 6. ■

We will refer to a variable or a value as an *atomic term.* For m-tuple $\mathbf{c}$, n-tuple $\mathbf{t}$ of atomic terms, n-tuple $\mathbf{d}$, and term s, we define the *composition* of entries $\mathbf{c} \to \mathbf{t}$ and $\mathbf{d} \to w$, written $(\mathbf{c} \to \mathbf{t}) \circ (\mathbf{d} \to w)$, as the entry returned by compose($\mathbf{c} \to \mathbf{t}, \mathbf{d} \to w$) shown in Figure 5.4. The composition of such entries is defined only when $\mathbf{c}$, $\mathbf{t}$, and $\mathbf{d}$ satisfy requirements that ensure the well-sortedness of the result.

Example 17 *$(*) \to (x_1, v_1) \circ (v_2, *) \to w$ is equal to $(v_2) \to w$.*

Example 18 *$(*, v_1) \to (x_2, x_1) \circ (v_1, v_2) \to w$ is equal to $(v_2, v_1) \to w$.*

Example 19 *$(*, v_2) \to (v_1, x_1) \circ (v_1, *) \to w$ is equal to $(*, v_2) \to w$.*

Lemma 8 *For m-tuple $\mathbf{c}$ and n-tuples $\mathbf{t}$ and $\mathbf{d}$, if $(\mathbf{c} \to \mathbf{t}) \circ (\mathbf{d} \to v)$ is the entry $\mathbf{w} \to v$, then $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{t}]$ in the theory of equality.*

Proof: By induction on n. If $n = 0$, then the method returns $\mathbf{c} \to v$, and since $\mathbf{d}$ is empty, $\gamma(\mathbf{c})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{t}]$. If $n > 0$, then if t_n is the variable x_j, then $\mathbf{c}.j$ is compatible with d_n, and by the induction hypothesis, the method returns $\mathbf{w} \to v$ such that $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c} \triangle_j d_n)[\mathbf{x}] \wedge \gamma((d_1, \ldots, d_{n-1}))[(t_1, \ldots, t_{n-1})]$. Since $\gamma(\mathbf{c} \triangle_j d_n)[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(d_n)[x_j]$, and t_n is x_j, we have that $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{t}]$. Otherwise, t_n is a value compatible with d_n, and by the induction hypothesis, the method returns $\mathbf{w} \to v$ such that $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma((d_1, \ldots, d_{n-1}))[(t_1, \ldots, t_{n-1})]$. Since t_n is a value compatible with d_n, we have that $\gamma(d_n)[t_n]$ is true, and thus $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{t}]$. ■

Definition 9 (Composition) *For definition $D_{\lambda\mathbf{x}.(t_1,\ldots,t_m)} = e_1, \ldots, e_i$, and definition $D_f = f_1, \ldots, f_j$ of arity m, we define their composition, written $D_{\lambda\mathbf{x}.f(t_1,\ldots,t_m)}$, as the definition $\emptyset \cdot e_1 \circ f_1 \ \cdot \ldots \ \cdot e_1 \circ f_j \ \cdot \ldots \ \cdot \ e_i \circ f_1 \ \cdot \ldots \ \cdot e_i \circ f_j$.*

We assume the obvious restriction that the composition of definitions $D_{\lambda\mathbf{x}.(t_1,\ldots,t_m)}$ and D_f is only defined when the composition of their entries is defined, according to requirements mentioned in Figure 5.4. Assuming our construction of D_f from the previous section, this means we will compute the composition of $D_{\lambda\mathbf{x}.(t_1,\ldots,t_m)}$ and D_f only when $f(t_1, \ldots, t_n)$ is a well-sorted term.

Example 20 *Say $D_{\lambda x_1 x_2.(t_1,t_2)}$ is $(*, *) \to (x_2, x_1)$ and D_f is $(v, w) \to v$, $(*, *) \to w$.*

Then $D_{\lambda x_1 x_2.f(t_1,t_2)}$ *is the definition* $(w,v) \to v$*,* $(*,*) \to w$*.*

Example 21 *Say* $D_{\lambda x_1.(t_1,t_2)}$ *is* $(v_1) \to (v_3, x_1)$*,* $(*) \to (v_2, x_1)$*, and* D_f *is* $(v_2, v_2) \to w_1$*,* $(*, v_1) \to w_2$*,* $(*,*) \to w_3$*. Then* $D_{\lambda x_1.f(t_1,t_2)}$ *is the definition* $(v_1) \to w_2$*,* $(v_2) \to w_1$*,* $(*) \to w_3$*.*

Lemma 9 *If* $D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)}$ *is a complete definition, and* D_f *is complete definition of arity* n*, then* (i) $D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)}$ *is a complete definition, and* (ii) $[\![\gamma(D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)})[\mathbf{v}]]\!] = [\![\gamma(D_f)[[\![\gamma(D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)})[\mathbf{v}]]\!]]]\!]$*, for all* $\mathbf{v}$*.*

Proof: Let $D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)}$ be $\mathbf{c_1} \to \mathbf{s_1}, \ldots, \mathbf{c_{m1}} \to \mathbf{s_{m1}}$ and D_f be $\mathbf{d_1} \to w_1, \ldots, \mathbf{d_{m2}} \to w_{m2}$. To show (i), since $D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)}$ and D_f are complete, then $\mathbf{c_{m1}}$ and $\mathbf{d_{m2}}$ are $*$. By Lemma 8, we have that $(\mathbf{c_{m1}} \to \mathbf{s_{m1}}) \circ (\mathbf{d_{m2}} \to w_{m2})$ is an entry $\mathbf{r} \to w_{m2}$ such that $\gamma(\mathbf{r})[\mathbf{x}] = \gamma(\mathbf{c_{m1}})[\mathbf{x}] \wedge \gamma(\mathbf{d_{m2}})[\mathbf{s_{m1}}] = \mathbf{true}$. Thus, $(\mathbf{c_{m1}} \to \mathbf{s_{m1}}) \circ (\mathbf{d_{m2}} \to w_{m2})$ is the entry $* \to w_{m2}$, and $D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)}$ is a complete definition.

To show (ii), let $\sigma = \{\mathbf{x} \mapsto \mathbf{v}\}$, and say $[\![\gamma(\mathbf{c_1})[\mathbf{x}]\sigma]\!], \ldots, [\![\gamma(\mathbf{c_{j-1}})[\mathbf{x}]\sigma]\!]$ are false, and $[\![\gamma(\mathbf{c_j})[\mathbf{x}]\sigma]\!]$ is true. Likewise, say $[\![\gamma(\mathbf{d_1})[\mathbf{s_j}]\sigma]\!], \ldots, [\![\gamma(\mathbf{d_{k-1}})[\mathbf{s_j}]\sigma]\!]$ are false, and $[\![\gamma(\mathbf{d_k})[\mathbf{s_j}]\sigma]\!]$ is true. First, for all $1 \leq i < j$, and all entries $\mathbf{d} \to w \in D_f$, say $(\mathbf{c_i} \to \mathbf{s_i}) \circ (\mathbf{d} \to w)$ is the entry $\mathbf{r_1} \to w$ for some $\mathbf{r_1}$. By Lemma 8, $[\![\gamma(\mathbf{r_1})[\mathbf{x}]\sigma]\!]$ is equal to $[\![(\gamma(\mathbf{c_i})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{s_i}])\sigma]\!]$, which is false since $[\![\gamma(\mathbf{c_i})[\mathbf{x}]\sigma]\!]$ is false. Second, for all $1 \leq i < k$, say $(\mathbf{c_j} \to \mathbf{s_j}) \circ (\mathbf{d_i} \to w)$ is the entry $\mathbf{r_2} \to w$ for some $\mathbf{r_2}$, By Lemma 8, $[\![\gamma(\mathbf{r_2})[\mathbf{x}]\sigma]\!]$ is equal to $[\![(\gamma(\mathbf{c_j})[\mathbf{x}] \wedge \gamma(\mathbf{d_i})[\mathbf{s_j}])\sigma]\!]$, which is false since $[\![\gamma(\mathbf{d_i})[\mathbf{s_j}]\sigma]\!]$ is false. Finally, say $(\mathbf{c_j} \to \mathbf{s_j}) \circ (\mathbf{d_k} \to w_k)$ is the entry $\mathbf{r_3} \to w_k$ for some $\mathbf{r_3}$. By Lemma 8, $[\![\gamma(\mathbf{r_3})[\mathbf{x}]\sigma]\!]$ is equal to $[\![\gamma(\mathbf{c_j})[\mathbf{x}] \wedge \gamma(\mathbf{d_k})[\mathbf{s_j}]\sigma]\!]$, which is true since $[\![\gamma(\mathbf{c_j})[\mathbf{x}]\sigma]\!]$ and

$[\![\gamma(\mathbf{d_k})[\mathbf{s_j}]\sigma]\!]$ are true. Thus, we have that $[\![\gamma(D_{\lambda\mathbf{x}.f(t_1,\ldots,t_n)})[\mathbf{v}]]\!] = w_k = [\![\gamma(D_f)[\mathbf{s_j}]\sigma]\!] = [\![\gamma(D_f)[[\![\gamma(D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)})[\mathbf{x}]\sigma]\!]]]\!] = [\![\gamma(D_f)[[\![\gamma(D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)})[\mathbf{v}]]\!]]]\!]$. ■

We may also apply interpreted functions to definitions whose range contains only values. In the following definition, we assume our evaluation operation $[\![t]\!]$ is extended to applications of built-in function symbols having sorts whose values in $\mathcal{M}$ are interpreted ground terms, such as the sorts Int and Bool.

Definition 10 (Interpreted Composition) *Given a definition $D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)} = \mathbf{c_1} \rightarrow \mathbf{t_1}, \ldots, \mathbf{c_m} \rightarrow \mathbf{t_m}$, the application of interpreted function or predicate f (of arity n) to $D_{\lambda\mathbf{x}.(t_1,\ldots,t_n)}$, written $D_{\lambda\mathbf{x}.f(t_1,\ldots,t_n)}$, is the definition $\mathbf{c_1} \rightarrow [\![f(\mathbf{t_1})]\!], \ldots, \mathbf{c_m} \rightarrow [\![f(\mathbf{t_m})]\!]$.*

Example 22 *Say $D_{\lambda x_1.(t_1,t_2)}$ is $(v) \rightarrow (1,5)$, $(*) \rightarrow (1,0)$, where 1, 5, and 0 are constants of sort Int. Then, $D_{\lambda x_1.+(t_1,t_2)}$ is the definition $(v) \rightarrow 6$, $(*) \rightarrow 1$, where $+$ is the built-in function symbol denoting addition.*

We assume interpreted composition of definitions can be computed for $\approx$ as well as other logical connectives by treating them as functions over terms of sort Bool, as shown in the following example.

Example 23 *Say $D_{\lambda x_1.(t_1,t_2)}$ is $(w) \rightarrow (v,v)$, $(*) \rightarrow (w,v)$. Then, $D_{\lambda x_1.\approx(t_1,t_2)}$ is the definition $(w) \rightarrow \mathbf{true}$, $(*) \rightarrow \mathbf{false}$.*

5.4.2.2 Computing Interpretations for Terms

Using the methods described in the previous section, we can compute $D_{\lambda\mathbf{x}.t}$ of a term t that is *model-checkable*, as defined in the following: [4]

[4] By this definition, a term is model-checkable if it is in the *essentially uninterpreted fragment*, as described in [28].

Definition 11 *A term t is model-checkable if and only if (i) t is a variable, (ii) t is $f(t_1, \ldots, t_n)$, f is uninterpreted and $t_1 \ldots t_n$ are model-checkable, or (iii) t is $f(t_1, \ldots, t_n)$, and $t_1 \ldots t_n$ are non-variable and model-checkable.*

To construct $D_{\lambda \mathbf{x}.t}$, in the case that t is a variable x_i, then $D_{\lambda \mathbf{x}.t}$ is the definition $* \to x_i$. In all other cases, $D_{\lambda \mathbf{x}.t}$ is computed bottom-up using the operations mentioned in the previous section. The following theorem states the correctness of our construction of $D_{\lambda \mathbf{x}.t}$ for a Σ-structure $\mathcal{M}$.

Theorem 5 *Let $\mathcal{M}$ be a Σ-structure induced by $\mathbf{D}_\Sigma$. For all model-checkable t, (i) $D_{\lambda \mathbf{x}.t}$ is a complete definition to atomic terms, and (ii) $[\![\gamma(D_{\lambda \mathbf{x}.t})[\mathbf{v}]]\!] = \mathcal{M}\sigma[\![t]\!]$, for all grounding substitutions $\sigma = \{\mathbf{x} \mapsto \mathbf{v}\}$.*

Proof: We show (*i*) and (*ii*) by induction on the structure of t.

Base case: We have that either t is a variable, or t is a constant. If t is the variable x_i, then $D_{\lambda \mathbf{x}.x_i}$ is the complete definition $* \to x_i$, and $[\![\gamma(D_{\lambda \mathbf{x}.t})[\mathbf{v}]]\!] = [\![\gamma(* \to x_i)[\mathbf{v}]]\!] = \mathbf{v}.i = x_i\sigma = \mathcal{M}\sigma[\![t]\!]$. Otherwise, say t is the constant $f()$. If f is uninterpreted, then D_f is $() \to w$ for some w. Thus, $D_{\lambda \mathbf{x}.f()}$ is the complete definition $(* \to ()) \circ (() \to w) = * \to w$, and $[\![\gamma(D_{\lambda \mathbf{x}.f()})[\mathbf{v}]]\!] = w = [\![\gamma(D_f)]\!] = \mathcal{M}\sigma[\![t]\!]$. If f is interpreted, then $D_{\lambda \mathbf{x}.t}$ is the complete definition $* \to [\![f()]\!]$, and $[\![\gamma(D_{\lambda \mathbf{x}.t})[\mathbf{v}]]\!] = [\![f()]\!] = \mathcal{M}\sigma[\![t]\!]$.

Inductive case: Assume t is $f(t_1, \ldots, t_n)$. To show (*i*), by the induction hypothesis and Lemma 7(*i*), $D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)}$ is a complete definition. When f is uninterpreted, since D_f is a complete definition, by Lemma 9(*i*), $D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)}$ is a complete definition. When f is interpreted, clearly $D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)}$ is a complete definition as well.

To show (ii), in the case that f is uninterpreted, then by Lemma 9(ii) we have that $[\![\gamma(D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)})[\mathbf{v}]]\!] = [\![\gamma(D_f)[[\![\gamma(D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)})[\mathbf{v}]]\!]]]\!]$, which by Lemma 7$(ii)$ is equal to $[\![\gamma(D_f)[([\![\gamma(D_{\lambda \mathbf{x}.t_1})[\mathbf{v}]]\!], \ldots, [\![\gamma(D_{\lambda \mathbf{x}.t_n})[\mathbf{v}]]\!])]]\!]$ which by the inductive hypothesis is equal to $[\![\gamma(D_f)[(\mathcal{M}\sigma[\![t_1]\!], \ldots, \mathcal{M}\sigma[\![t_n]\!])]]\!]$, which, since $\mathcal{M}$ is induced by $\mathbf{D}_\Sigma$, is equal to $\mathcal{M}\sigma[\![f(t_1, \ldots t_n)]\!]$. Otherwise if f is interpreted, let $\mathbf{c} \to \mathbf{v}$ be the first entry in $D_{\lambda \mathbf{x}.(t_1,\ldots,t_n)}$ for which $[\![\gamma(\mathbf{c})[\mathbf{v}]]\!]$ is true. By the induction hypothesis, and since t is model-checkable, $\mathbf{v}$ must be the n-tuple of values $(\mathcal{M}\sigma[\![t_1]\!], \ldots, \mathcal{M}\sigma[\![t_n]\!])$. We have $[\![\gamma(D_{\lambda \mathbf{x}.f(t_1,\ldots,t_n)})[\mathbf{v}]]\!] = [\![f(\mathbf{v})]\!] = [\![f(\mathcal{M}\sigma[\![t_1]\!], \ldots, \mathcal{M}\sigma[\![t_n]\!])]\!] = \mathcal{M}\sigma[\![f(t_1, \ldots, t_n)]\!]$.

■

Extensions In some cases, we may compute the interpretation of terms that are not model-checkable. First, we can compute the interpretation of terms whose free variables are of sorts with finite domains, regardless of whether they contain variables that occur as children of interpreted symbols. The idea is that we can always interpret the variable as an explicit enumeration of its corresponding domain elements. So, for a quantifier over one variable x_1 whose sort has domain elements $\{v_1, v_2, v_3\}$, we may construct $D_{\lambda x_1.x_1}$ as $(v_1) \to v_1$, $(v_2) \to v_2$, $* \to v_3$. By doing so, we may subsequently apply interpreted functions to $D_{\lambda x_1.x_1}$.

We can also compute interpretation for equalities with exactly one variable child. If $D_{\lambda \mathbf{x}.t}$ is definition $\mathbf{c_1} \to v_1, \ldots, \mathbf{c_n} \to v_n$, then $D_{\lambda \mathbf{x}.t \approx x_i}$ (likewise $D_{\lambda \mathbf{x}.x_i \approx t}$) is equal to $\mathbf{c_1} \triangle_i v_1 \to \mathbf{true} \cdot \mathbf{c_1} \to \mathbf{false} \cdot \ldots \cdot \mathbf{c_n} \triangle_i v_n \to \mathbf{true} \cdot \mathbf{c_n} \to \mathbf{false}$.

Example 24 *Say* $D_{\lambda x_1 x_2.t}$ *is* $(*, v_3) \to v_4$, $(v_1, v_0) \to v_1$, $(*, *) \to v_2$. *Then,* $D_{\lambda x_1 x_2.t \approx x_1}$ *is* $(v_4, v_3) \to \mathbf{true}$, $(*, v_3) \to \mathbf{false}$, $(v_1, v_0) \to \mathbf{true}$, $(v_2, *) \to \mathbf{true}$, $(*, *) \to \mathbf{false}$.

Example 25 *Say* x_1 *and* x_2 *are variables whose sort has domain* $\{v_1, v_2, v_3\}$. *Then,* $D_{\lambda x_1\, x_2. x_1 \approx x_2}$ *is equal to* $(v_1, v_1) \rightarrow$ **true**, $(v_2, v_2) \rightarrow$ **true**, $(v_3, v_3) \rightarrow$ **true**, $(*, *) \rightarrow$ **false**. [5]

5.4.2.3 Choosing Instantiations

Constructing interpretations for terms allows us to check the satisfiability of quantified formulas with respect to a Σ-structure $\mathcal{M}$ representing a candidate model. Consider a quantified formula $\forall \mathbf{x}.(\varphi \vee \psi)$ where φ is model-checkable. If $D_{\lambda \mathbf{x}.\varphi}$ contains no entries of the form $\mathbf{c} \rightarrow$ **false**, then using Theorem 5, it can be shown that $\mathcal{M}$ satisfies $\forall \mathbf{x}.(\varphi \vee \psi)$. Notice that this does not depend on the fact that the sorts of $\mathbf{x}$ are finite.

Our procedure for choosing substitutions $I_{\mathbf{x}}$ for a quantified formula $\forall \mathbf{x}.\varphi$ will be based on the entries of the form $\mathbf{c} \rightarrow$ **false** in $D_{\lambda \mathbf{x}.\varphi}$. For each such entry, we will add at most one substitution to $I_{\mathbf{x}}$. Since the simplification techniques from Section 5.3.4 are incomplete, entries of this form may be *unreachable*, that is, it can be the case that $\mathbf{c}$ does not generalize any tuple of ground values that evaluates to **false** in $D_{\lambda \mathbf{x}.\varphi}$. Thus, our method will, in the worst case, search for such a tuple using the methods described in the previous section.

The following describes the instances chosen by our heuristic $\mathcal{H}$ for a quantified formula $\forall \mathbf{x}.\varphi$, given a candidate model $\mathcal{M}$.

---

[5] In the implementation, equality between variables (with domain size n) is handled as a special case to avoid computing a product containing n^2 entries prior to simplification.

$\mathcal{H}$ for $\forall\mathbf{x}.\varphi$: Compute $D_{\lambda\mathbf{x}.\varphi}$ using the methods mentioned in this section (since the domain of $\mathbf{x}$ is finite in $\mathcal{M}$, we may always compute $D_{\lambda\mathbf{x}.\varphi}$, regardless of whether φ is model-checkable or not). For each $\mathbf{c} \rightarrow \mathbf{false} \in D_{\lambda\mathbf{x}.\varphi}$, let σ be the substitution mapping x_i to $\mathbf{c}.i$ for all i such that $\mathbf{c}.i$ is not $*$. Call choose_instances on $\mathcal{M}$ and $\varphi\sigma$, which returns a set of substitutions J with domain $\mathbf{x} \setminus \mathcal{D}om(\sigma)$. If J is non-empty, add $\sigma \cdot \sigma'$ to $I_{\mathbf{x}}$ for some $\sigma' \in J$.

Proposition 3 *$\mathcal{H}$ returns an empty set of instantiations if and only if $\mathcal{M} \models \forall\mathbf{x}.\varphi$.*

Proof: If $\mathcal{M}\sigma[\![\varphi]\!]$ is false for some $\sigma = \{\mathbf{x} \mapsto \mathbf{v}\}$, by Theorem 5, $[\![\gamma(D_{\lambda\mathbf{x}.\varphi})[\mathbf{v}]]\!]$ is false, and thus $\mathbf{c} \rightarrow \mathbf{false}$ is an entry in $D_{\lambda\mathbf{x}.\varphi}$ for some $\mathbf{c}$ such that $[\![\gamma(\mathbf{c})[\mathbf{v}]]\!]$ is true. We will call choose_instances on $\varphi\sigma'$, where $\varphi\sigma$ is an instance of $\varphi\sigma'$, and hence by Proposition 2 we will add at least one instance in this case. Conversely, if $\mathcal{M}\sigma[\![\varphi]\!]$ is true for all σ, we apply choose_instances in the most general case to φ itself, which by Proposition 2 is guaranteed to produce no instances. ■

Using this quantifier instantiation heuristic $\mathcal{H}$, we may significantly reduce the domain size for which the choose_instances procedure is run, as compared to the previous section, since it is being called on partially instantiated version of φ. This is shown in the following example.

Example 26 *Consider the quantified formula $Q = \{\forall x_1\, x_2.\, \varphi\}$, where $\varphi = f(x_2) \approx g(x_1, b) \vee h(x_1, x_2) \not\approx b$, and function definitions from Example 13. For the left disjunct of φ, we get $D_{\lambda x_1\, x_2. f(x_2) \approx g(x_1, b)} = (*, b) \rightarrow$ **false**$, (*, *) \rightarrow$ **true**. For the right disjunct,*

we get $D_{\lambda x_1\, x_2. h(x_1,x_2)\not\approx b} = (*, *) \rightarrow \textbf{false}$. *Overall, our definition* $D_{\lambda x_1\, x_2. \varphi}$ *is* $(*, b) \rightarrow \textbf{false}, (*, *) \rightarrow \textbf{true}$. *We then find a substitution* $\{x_1 \mapsto v\}$ *for some* v *such that* $\varphi\{x_2 \mapsto b\}$ *is falsified, for which we add* $\{x_1 \mapsto v, x_2 \mapsto b\}$ *to* $I_{\mathbf{x}}$. ■

Notice that in Example 13, the algorithm from Section 5.4.1 concluded that all instances of the form $\varphi\{x_2 \mapsto a\}$ are satisfied by $\mathcal{M}$. In light of Example 26, we can see that all instances of form $\varphi\{x_2 \mapsto w\}$ for any value $w \neq b$ are satisfied by $\mathcal{M}$. Whereas the previous algorithm would have concluded this independently for each such w, the algorithm in this section avoids this repeated computation.

Optimized Heuristic for Model Checkable Formulas In many cases, determining the set of substitutions $I_{\mathbf{x}}$ from $D_{\lambda \mathbf{x}. \varphi}$ can be done immediately based on the following observation. We say an entry $\mathbf{c} \rightarrow v$ is *pure with respect to distinguished values* if and only if $\mathbf{c}$ does not contain any occurrence of $\mathcal{A}_M(e^S)$ for any sort S. If each entry in $D_{\lambda \mathbf{x}. \varphi}$ is pure with respect to distinguished values, then if $\mathbf{c} \rightarrow \textbf{false}$ in $D_{\lambda \mathbf{x}. \varphi}$, then clearly replacing all occurrences of $*$ with $\mathcal{A}_M(e^S)$ in $\mathbf{c}$ results in a tuple of values that evaluates to **false** in $D_{\lambda \mathbf{x}. \varphi}$, and hence can be used for finding a falsified instance when constructing $I_{\mathbf{x}}$.

(Optimized) $\mathcal{H}$ for $\forall \mathbf{x}. \varphi$: Compute $D_{\lambda \mathbf{x}. \varphi}$. For each $\mathbf{c} \rightarrow \textbf{false} \in D_{\lambda \mathbf{x}. \varphi}$, let σ be the substitution mapping x_i to $\mathbf{c}.i$ for all i such that $\mathbf{c}.i$ is not $*$. and x_i to $\mathcal{A}_M(e^S)$ for all i such that $\mathbf{c}.i$ is $*$. Add the substitution σ to $I_{\mathbf{x}}$.

$$\forall\text{-}\textbf{Inst}\ \frac{a \in \mathsf{M} \quad a \Leftrightarrow \forall \mathbf{x}\, \varphi \in \mathsf{F}}{\mathsf{F} := \mathsf{F} \cup \{\neg a \vee \varphi\{\mathbf{x} \mapsto \mathbf{t}\}\}}$$

Figure 5.5. DPLL($T_1, \ldots, T_m$) rule for quantifier instantiation.

When φ is model-checkable and $\mathcal{M}$ is a fragmented model [6], then $D_{\lambda \mathbf{x}.\varphi}$ is pure with respect to distinguished values. This can be shown by noting that all function definitions in fragmented models $\mathcal{M}$ are pure with respect to distinguished values, and all basic operations on definitions (product, compose, interpreted compose) preserve this property. In this case, we use the optimized version of our quantifier instantiation heuristic described above.

In the implementation, we call the optimized $\mathcal{H}$ on all quantified formulas first. If this produces no instances and $\mathcal{M}$ is a fragmented model, we call the non-optimized version of $\mathcal{H}$ on all quantified formulas whose bodies are not model-checkable.

5.4.3 Integration into DPLL($T_1, \ldots, T_m$)

To integrate our quantifier instantiation heuristics into the DPLL($T_1, \ldots, T_m$) architecture, we extend it with the rule shown in Figure 5.5. When a, which serves as a proxy for the quantified formula $\forall \mathbf{x}\, \varphi$ as described in Appendix A, occurs positively in the current assignment M, the SMT solver may add ground instances of the formula φ to F using the rule $\forall$-**Inst**. We assume that all preprocessing techniques from Appendix A are applied to $\varphi\{\mathbf{x} \mapsto \mathbf{t}\}$, and thus one application of this rule may result in adding multiple clauses to F. For termination of DPLL($T_1, \ldots, T_m$) with

[6]See Section 5.3.3.

quantifier instantiation, we require that the terms **t** are chosen from a finite set, and that executions contain no redundant applications of $\forall$-**Inst**.

5.5 Properties

We now prove several important properties of our approach for finite model finding in SMT. We assume the model finding procedure introduced in Section 5.1 is used, and that fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) is used for finding satisfying assignments for ground clauses in Step 1. In the following, we will refer to the overall approach as *fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) with quantifier instantiation*, which is applicable to sets of clauses F_0 whose quantification is limited to uninterpreted sorts.

First, we remark that our approach is sound, due to Theorem 3 and that all clauses added by quantifier instantiation preserve the satisfiability of F_0. Second, when fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) with quantifier instantiation terminates with model $\mathcal{M}$, then indeed $\mathcal{M} \models F_0$. This is a consequence of Theorem 3, the correctness of our preprocessing techniques for existential quantifiers, as well as the correctness of our quantifier instantiation heuristics for universal quantifiers, e.g. Propositions 2 and 3.

5.5.1 Finite Model Completeness

Theorem 6 *Given a signature Σ with uninterpreted sorts $S_1, \ldots, S_n$, fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) with quantifier instantiation is finite model complete for every set of Σ-clauses F_0 whose quantification is limited to $S_1, \ldots, S_n$.*

Proof: Assume that F_0 has a model $\mathcal{M}$ in which $S_1, \ldots S_n$ have finite domain sizes

$k_1, \ldots, k_n$. Let Q be the set of all quantified formulas in F_0, and assume that all nested quantification in Q has been removed using the methods described in Appendix A.4, Thus, no quantified formulas beyond those in Q are introduced during our procedure. At any time during the procedure, as a consequence of Proposition 1, the cardinality of any sort S_i in a candidate model will be at most $(k_1 + \ldots + k_n) - n$. Furthermore, due to our selection of representatives as described in Section 5.3.1, our procedure will instantiate all quantified formulas in Q with terms t from Σ where $\mathsf{depth}(t)$ is at most $(k_1 + \ldots + k_n) - n$.

For a quantified formula $\varphi \in Q$, let J_φ be the set of instances of the form $\varphi\sigma$, where σ is a substitution mapping variables to terms t such that $\mathsf{depth}(t)$ is at most $(k_1 + \ldots + k_n) - n$. Since fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) is terminating for any set of ground clauses due to Theorem 3, and since the instantiations we consider (the set $\bigcup_{\varphi \in Q} J_\varphi$) is finite, we will execute only a finite number of instantiation rounds (Step 3 of Definition 1) before terminating with a model. ■

5.5.2 Refutational Completeness

We also show refutational completeness of our approach, which however we will restrict to the case where no background theories besides EUF are present, and a naive approach for quantifier instantiation is used. Showing refutational completeness in the presence of background theories and for other quantifier instantiation heuristics is left for future work.

Theorem 7 *Fixed-cardinality DPLL(T_{EFCC}) with naive quantifier instantiation is refutationally complete for every set of Σ-clauses F_0.*

Proof: Assume that F_0 is unsatisfiable. Let Q be the set of quantified formulas occurring in F_0, where for simplicity we assume that nested quantification has been removed from each quantified formula. As a consequence of Herbrand's theorem and the compactness theorem for first-order logic, there exists a finite set of ground instances J of Q that together with F_0 is unsatisfiable. Let S be the set of all Σ-terms which were used to instantiate variables of Q in J, and let k be the smallest integer such that $\mathsf{depth}(t) \leq k$ for all $t \in S$. Let J^* be the set of all instances of the form $\varphi\sigma$, where $\varphi \in Q$ and σ is a substitution mapping variables to terms t such that $\mathsf{depth}(t)$ is at most k. Since k is finite, we know that J^* is finite, and since $J \subseteq J^*$, we know that $F_0 \wedge J^*$ is unsatisfiable.

On any instantiation round, the naive quantifier instantiation heuristic $\mathcal{H}$ will add at least one instance from J^* to our current set of clauses F. To show this is the case, say we have constructed a candidate model $\mathcal{M}$. Since $J^* \wedge F$ is unsatisfiable, there must be at least one instance $\varphi\sigma$ from J^* that is not satisfied by $\mathcal{M}$. Let σ' be a substitution mapping each variable x in the domain of sigma to the term from $\mathbf{V}$ (the domain elements of $\mathcal{M}$) that is equivalent to $x\sigma$ in $\mathcal{M}$. Due to our selection of $\mathbf{V}$ which chooses representative terms with minimal depth, and since $\varphi\sigma \in J^*$, we know that $\varphi\sigma' \in J^*$ as well. Since $\varphi\sigma'$ is also not satisfied by $\mathcal{M}$, we know that the instance $\varphi\sigma'$ has not yet been added to F. Since the naive quantifier instantiation heuristic $\mathcal{H}$ chooses all instances based on the domain of the model that have not yet been added to F, it will add $\varphi\sigma'$ to F, and thus the instantiation round adds at least one instance from J^* to F.

Since J^* is a finite set, and since no execution of fixed-cardinality DPLL(T_{EFCC}) is non-terminating between instantiation rounds due to Theorem 3, our procedure, in the worst case, will eventually add all instances from J^* to our set of clauses F, where $F_0 \subseteq F$. When this is the case, since $J^* \wedge F_0$ is unsatisfiable, and since fixed-cardinality DPLL($T_1, \ldots, T_m, T_{\text{EFCC}}$) is complete due to Theorem 3, the procedure will terminate, answering unsatisfiable. ■

In this proof, we limit ourselves to the case where no background theories occur, so that the compactness theorem for first-order logic can be trivially applied. We would like to extend this result to prove the refutational completeness of our approach for inputs F_0 containing background theories, but where quantification in F_0 is limited to uninterpreted sorts. We believe a similar argument as the one above can be used for restricted cases of this form, but this is left as future work. Regardless, given an argument of the form above, our approach has a weaker yet still useful property for any unsatisfiable formula F_0 containing background theories whose quantification is limited to uninterpreted sorts. Namely, if a finite number of instantiations of quantified formulas from F_0 suffices to show F_0 is unsatisfiable, our procedure will terminate with the answer unsatisfiable. An example of when this is not the case is demonstrated in the following example, where our algorithm does not terminate.

Example 27 *Say we wish to determine the satisfiability of the clauses* $\{\forall x. f(succ(x)) \approx f(x) + 1, \forall x. P(x) \Leftrightarrow \neg P(succ(x)), P(a), P(b), f(a) \approx f(b) + 2 * k + 1\}$, *in the combined theory of* EUF *and linear integer arithmetic, where the signature contains the integer sort* Int, $P : S \rightarrow \mathsf{Bool}$, $f : S \rightarrow \mathsf{Int}$, $succ : S \rightarrow S$, $a, b : S$, *and* $k : \mathsf{Int}$. *Con-*

ceptually, f is an injection mapping S to the integers, succ is a successor function on S with respect to this mapping, and P is true for every other element of S. This formula is unsatisfiable since P holds for both a and b, and a and b are separated by odd number of elements due to the last clause. However, this set is satisfiable for any finite number of quantifier instantiations since the value of k can be made arbitrarily large.

Given the two properties mentioned in this section, for inputs F_0 whose quantification is limited to uninterpreted sorts, our procedure is only non-terminating when F_0 is satisfiable but has no finite models, or when F_0 is unsatisfiable but any finite subset of instances of formulas from F_0 is satisfiable. In the first case, we say F_0 is satisfiable but *finitely unsatisfiable*. Recent work [14] has focused on automatically determining when a formula is finitely unsatisfiable using various techniques, such as determining the existence of an automorphism that is injective but not surjective, as shown in the following example.

Example 28 *Say we wish to determine the satisfiability of the clauses $\{\forall xy.f(x) \approx f(y) \Rightarrow x \approx y, \forall x.f(x) \not\approx a\}$ where all terms are of sort S. If this set had a model of some finite size k, then the injective function f must map the k elements of sort S collectively to each of the k elements of sort S. However, this is impossible since there is at least one element, namely a, that is not mapped to by f. Thus, this set of clauses is finitely unsatisfiable, but however has models of infinite size.*

For this example, our approach will consider finite models of larger and larger size, and thus will not terminate. Coupling techniques for recognizing infinite models

with the techniques described in this thesis is left for future work.

5.6 Enhancements

In this section, we mention several enhancements that can improve performance for our approach to finite model finding in SMT.

5.6.1 Heuristic Instantiation

As mentioned, many SMT solvers rely on heuristic instantiation methods for finding unsatisfiable instances for quantified formulas. We found that these methods can be helpful in our model finder as well, even for satisfiable problems, most likely because the instances it generates are helpful in pruning the search space. Following the terminology used in this chapter, our original heuristic $\mathcal{H}$ for quantifier instantiation can be enhanced with E-matching to a heuristic $\mathcal{H}$ as follows.

1. Choose a set of patterns T_ψ for each $\psi \in Q$, and return substitutions based on E-matching for (T_ψ, F).

2. If no such substitutions exist, apply the original $\mathcal{H}$.

Applying E-matching helps the model finder detect the unsatisfiability of its input formulas more promptly in cases where a conflict is easily identifiable. Furthermore, it may accelerate the search for finding models, since the instances it generates can help rule out candidate models more quickly.

In our approach, quantifier instantiation is applied *after* finding a satisfying assignment with a bounded number of equivalence classes. By waiting to apply quantifier instantiation until after a satisfying assignment of this form can be constructed,

we can avoid pitfalls common to E-matching-based procedures, such as matching loops. Since only a finite number of terms will be considered for a given cardinality bound on a sort, our approach guarantees that E-matching will eventually rule out the current cardinality bound, or terminate with no instances produced.

5.6.2 Sort Inference

When searching for models, it is often critical to reduce symmetries inherent to the problem. Informally, symmetries occur when a formula is satisfied by a multiple permutations of values. One major source of symmetry reduction can be discovered when applying *sort inference* to an input problem, which effectively limits the number of models the solver needs to search for. Various automated theorem provers take advantage of information regarding sort inference [16, 40]. This section gives a brief overview of two approaches where sort inference can be leveraged for our approach to finite model finding in SMT.

First, we introduce a basic sort inference technique used by several recent finite model finding tools, and how they are typically used. Given a signature Σ and Σ-formula φ, we will construct a signature Σ_i, with the following properties. As before, we assume that we have a separate equality symbol in Σ_i for each sort. For each interpreted function or predicate $f \in \Sigma$ other than equality, we have that f is also in Σ_i. For each uninterpreted function or predicate $f \in \Sigma$ of arity n, there exists a function of the same arity in Σ, but possibly with a different sort. For convenience, we will assume this function has the same name as f. Similarly, for each variable x in Σ, we associate a variable in Σ_i of the same name but possibly having a different

sort than the sort of x in Σ. We will say that Σ_i is an *inferred signature* for φ if φ remains well-sorted when viewed as a Σ_i-term. For a sort S, we will say S_j is an *inferred subsort* of S (according to Σ_i) if there exists a term t that has sort S in Σ, and sort S_j in Σ_i.

For a Σ-formula φ, we wish to construct a *maximally diverse* signature Σ_i such that φ is well-sorted according to Σ_i, that is, a signature having a maximal number of inferred subsorts. To find such a signature Σ_i, we first introduce a unique sort for each argument of functions in our signature, as well as a unique sort for the return value for functions and variables in our signature. Then we perform a single traversal over φ. Using a union-find data structure for storing an equivalence relation between sorts, we merge classes of sorts that must be equal to ensure that φ is well-sorted according to Σ_i.

As mentioned, in the approach used by several ATP finite model finders, the finite satisfiability of first-order quantified formulas can be reduced to the Boolean satisfiability problem. This reduction relies on introducing a set of constants for representing the domain elements of a model of finite cardinality, and various other constraints for encoding function symbols. Say we are searching for models of cardinality k for a sort S, and $c_1, \ldots, c_k$ are the domain constants associated with sort S. All terms of sort S must be equal to one of $c_1, \ldots, c_k$. For terms $t_1, \ldots, t_n$ of sort S, symmetries may be reduced by adding clauses to the underlying SAT solver corresponding to the encoding of:

$$(t_1 \approx c_1) \wedge (t_2 \approx c_1 \vee t_2 \approx c_2) \wedge \ldots \wedge (t_{k-1} \approx c_1 \vee \ldots \vee t_{k-1} \approx c_{k-1}) \qquad (5.12)$$

Additional clauses can enforce the canonicity of the models found by this approach, in particular by saying that t_i can be equal to c_j only when t_{i-1} is equal to c_{j-1}. Information regarding inferred subsorts can strengthen these clauses even further. Say we have an inferred signature Σ_i for φ, and $t_1, \ldots, t_m$ are of sort S_1 in Σ_i and $t_{m+1}, \ldots, t_n$ are of sort S_2 in Σ_i. It can be shown that a model exists interpreting these sets of terms as separate sorts if and only if a model exists interpreting them as the same sort, and hence symmetry reduction clauses of the form 5.12 can be added for both of these sets separately.

Symmetry Reduction Clauses in SMT Recall that our approach for finite model finding does not rely on the introduction of domain constants, and instead enforces cardinality constraints explicitly through a specialized theory solver for EUF with finite cardinality constraints. Since these constants are not introduced, there is no need to add clauses of the form shown in 5.12 for terms of the same sort. However, sort inference information can be used to break other symmetries, using the following approach.

Say we are given an input Σ-formula φ that is well-sorted according to an inferred signature Σ_i. The approach mentioned in this section will still treat φ as a Σ-formula, but use the sort information from Σ_i to add additional clauses to the solver for the purposes of breaking symmetries.

Assume we have a total ordering on Σ-terms $\succeq$. Given ground terms t_n and s_m having sort S in Σ and distinct sorts in Σ_i, let $t_1, \ldots, t_{n-1}$ and $s_1, \ldots, s_{m-1}$ be the set of all Σ-terms having the same inferred subsort according to Σ_i that are less than

t_n and s_m respectively according to $\succeq$. Then, we may add the clause:

$$(t_n \approx s_1 \vee \ldots \vee t_n \approx s_{m-1} \vee s_m \approx t_1 \vee \ldots \vee s_m \approx t_{n-1} \vee t_n \approx s_m) \quad (5.13)$$

When read as an implication, this says that if t_n is disequal from all terms smaller than s_m, and likewise s_m is disequal from all terms smaller than t_n, we have that $t_n \approx s_m$. This can be assumed without loss of generality for the same reasoning that symmetry reduction can be applied to each inferred subsort separately in the previous section. The difference in our approach is that these clauses may be added *on demand* throughout the procedure for terms t_n and s_m for which we determine a clause of this form is necessary.

In the implementation, the ordering $\succeq$ is chosen dynamically during the procedure. That is, we maintain an (initially empty) vector of terms $U = [t_1, \ldots, t_n]$, for which we say that $t_n \succeq \ldots \succeq t_1$. Say we are searching for models of size k for some sort S. For each inferred sub-sort S_i of S, we maintain a context-dependent vector U_{Si} of at most k terms of that sort that are pairwise entailed to be disequal in the current assignment. When a ground term t of inferred sort S_i becomes disequal from every term in U_{Si}, we append it to U_{Si}, and also append it to U if it does not already exist in U. [7] We add clauses of the form in 5.13 for two distinct inferred subsorts S_i and S_j whenever a term t is added to U_{Si}, and the vector U_{Sj} associated with S_j has maximal length among all inferred subsorts of S. In particular, we add the clause for t and its counterpart at the same index in U_{Sj}.

[7] For each inferred subsort S_i, we keep track of disequalities from U_{Si} to single watched term t that is not entailed to be equal to any term in U_{Si}. If there exists any term $t' \in U \backslash U_{Si}$, then t is the first such term in the vector U.

Using the Inferred Signature As a more direct alternative to that mentioned in the previous section, to determine the satisfiability of the Σ-formula φ, we may determine the satisfiability of the Σ_i-formula $\varphi_i \wedge C_{mon}$, where Σ_i is an inferred signature for φ, and φ_i is syntactically identical to φ but whose subterms may have different sorts, and C_{mon} are additional constraints based on the *monotonicity* of φ with respect to our inferred subsorts, as described below.

Definition 12 *A formula φ is* monotonic *with respect to sort S if and only if whenever there exists a model $\mathcal{M}_1$ for φ with domain $\mathbf{V}_1^S$ for S, then there exists a model $\mathcal{M}_2$ for φ with domain $\mathbf{V}_2^S$ for S, where $\mid \mathbf{V}_2^S \mid = \mid \mathbf{V}_1^S \mid +1$.*

For example, consider the formula φ of the form $(\forall xy.x \approx y) \wedge t \not\approx s$, where t, s, x, and y have sort S. This formula is unsatisfiable, since the first conjunct, $(\forall xy.x \approx y)$, only has models of size 1 for S, and the second conjunct only has models where the size of S is greater than 1. Running our sort inference algorithm on φ would assign a sort S_1 for x and y and another sort S_2 for t and s. Let φ_i be the corresponding Σ_i-formula for φ. We can see that φ_i is monotonic with respect to S_2 but not to S_1, and is satisfiable with a model where $\mid \mathbf{V}^{S1} \mid = 1$ and $\mid \mathbf{V}^{S2} \mid = 2$. However, if we enforce that the cardinality of S_1 is at least as large as S_2, then φ_i is unsatisfiable. As described earlier, we construct an additional constraint C_{mon} for doing so, namely by introducing an injective function from S_2 to S_1.

Following the approach mentioned in [15], when checking the satisfiability of φ in an inferred signature, if φ is not monotonic with respect to an inferred subsort S_j of sort S, then all other inferred subsorts of S must have cardinality less than or

equal to S_j. Thus, our approach for constructing C_{mon} is as follows. For each sort S in Σ, if there exists a inferred subsort S_j of S in Σ_i for which φ is not monotonic, then for all other inferred subsorts S_k of S, we introduce a fresh function f of type $S_k \rightarrow S_j$, and add the formula $\forall xy.f(x) \approx f(y) \rightarrow x \approx y$ as a conjuct of C_{mon} to express that f is injective. If φ is also not monotonic with respect to S_k, we also introduce an injective function from S_j to S_k.

With respect to the method described in the previous section, one disadvantage of this approach is that defining injective functions in C_{mon} introduces quantified formulas. However, in many use cases, φ contains quantified formulas already, and thus the overhead here is often negligible. Another disadvantage is that the introduction of multiple sorts makes it more difficult to enforce a fair strategy when incrementing cardinalities, as described in Section 5.2.4. On the other hand, the advantage of the approach is that no clauses of the form described in 5.13 are needed for reducing symmetries, and hence less bookkeeping is required during the search. Another major advantage of this approach is that we may reduce the number of ground instances for formulas containing variables with subsorts S_i of S. This occurs when a model can be found that interprets S_i as a smaller set than the interpretation of the other inferred subsorts of S.

We infer monotonicity for formulas and sorts using incomplete methods, since unfortunately, determining the monotonicity of a formula with respect to a sort is in general an undecidable problem. Recently, several calculi have been proposed [15, 9] for recognizing cases when it is possible to infer monotonicity, including the approach

of Monotonox [15], which encodes an incomplete check for inferring monotonicity with respect to sorts in an input problem as a satisfiability problem. These approaches recognize when a formula φ is not monotonic with respect to a sort S based on the structure of the formula. In particular, when φ contains a variable of sort S that occurs as a direct child of an equality having positive polarity, then φ is not guaranteed to be monotonic with respect to S. Like other approaches, when we cannot determine that a formula is monotonic with respect to a sort, we assume that it is not.

5.6.3 Relevancy

Several SMT solvers, including cvc4, use heuristics for reducing the size of satisfying assignments during the DPLL($T_1, \ldots, T_m$) search. These heuristics are based on including only literals that contribute to satisfying the given set of clauses [18]. For model-based quantifier instantiation, these techniques help performance considerably, since minimizing the number of literals in the satisfying assignment reduces the number of terms in its corresponding congruence closure. This leads to candidate models that have fewer entries in function definitions, and thus are easier to check, and are less likely to contain entries that falsify instances of quantified formulas.

5.7 Results

We implemented all features mentioned in this chapter into cvc4 [3], a state-of-the-art SMT solver based on the DPLL($T_1, \ldots, T_m$) architecture. This section presents experimental results on this implementation. We separate this section into two sets of experiments, the first to evaluate the relative effectiveness of various

strategies for the EFCC solver, and the second to evaluate the model finder's overall performance when used with quantified formulas. For the second set of experiments, we compare our model finder against state-of-the-art SMT solvers and automated theorem provers.

5.7.1 EFCC Solver Evaluation

We first examine the effectiveness of approach to handling ground problems in the theory of EUF with finite cardinality constraints (EFCC). In this section, all experiments were run on a Linux machine with an 8-core 2.60GHz Intel® Xeon® E5-2670 processor with 16GB of RAM.

We tested various configurations of the EFCC solver, starting with the default configuration **cvc4+f**, which contains the region-based enhancements described in Section 5.2.3, where conflicting states are reported by using clique lemmas of the form $\neg\mathsf{distinct}(c_1, \ldots, c_k) \vee \neg\mathsf{card}_{S,k}$. We also tested a configuration, **cvc4+fe**, where conflict clauses are as described in Section 5.2.2. This configuration avoids the introduction of new equalities into the search (contained in the expansion of $\mathsf{distinct}$), but has the disadvantage that it can generate different conflict clauses for essentially the same clique. Additionally, we considered configuration **cvc4+f-r**, which differs from **cvc4+f** only in that regionalizations have always just one region per sort S, encompassing the entire disequality graph for S.

We also evaluated the MACE-style approach to finite model finding described in related work, which we encoded in the configuration **cvc4+mace**. For a basic idea of this encoding in the simple case of a set of ground clauses F involving a single sort,

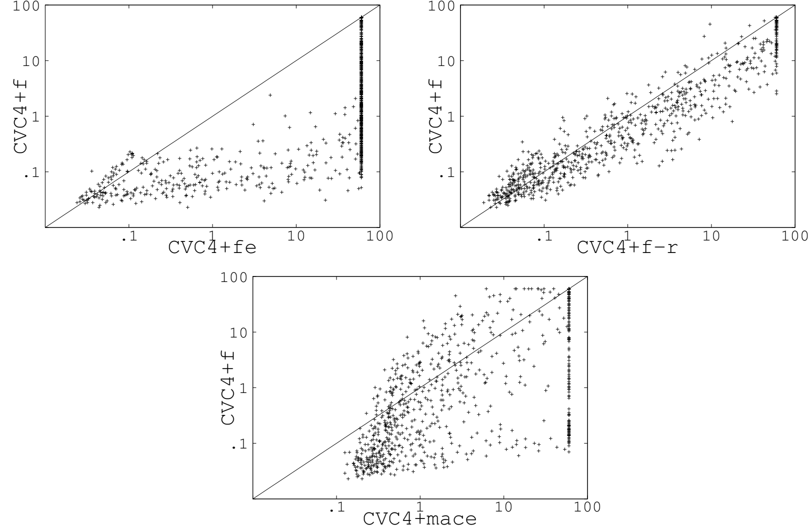

Figure 5.6. Results for randomly generated benchmarks. Runtimes are on a log-log scale.

if $\mathbf{T}_F$ is the set of all terms in F and $c_1, \ldots, c_k$ are fresh constants serving as domain constants, this configuration uses CVC4 to check the satisfiability of

$$F \wedge \mathsf{distinct}(c_1, \ldots, c_k) \wedge \bigwedge_{t \in \mathbf{T}_F} (t \approx c_1 \vee \ldots \vee t \approx c_k) \tag{5.14}$$

for $k = 1, 2, \ldots$ until (5.14) is found satisfiable for some k. Then, the minimal model size for F is k. As mentioned, a major shortcoming of this approach is the introduction of unwanted symmetries in the problem. CVC4 can address this issue to some extent since it incorporates symmetry breaking techniques directly at the ground EUF level [22].

We considered satisfiable benchmarks encoding randomly generated graph coloring problems and consisting of a conjunction of disequalities between constants of a single sort. In particular, we considered a total of 793 non-trivial problems containing between 20 and 50 unique constants and between 100 and 900 disequalities,

and measured the time it takes each configuration to find a model of minimum size, with a 60 second timeout. For the benchmarks we tested, the configuration **cvc4+f** solves the most benchmarks within the time limit: 723. The configuration **cvc4+f** was an order of magnitude faster than **cvc4+fe** on most benchmarks, with the latter only being able to solve 309 benchmarks within the time limit. This strongly suggests that generating explanations for cliques in conflict lemmas involving cardinality constraints is not an effective approach in this scheme.

Figure 5.6 compares the performance of the configuration **cvc4+f** against **cvc4+fe**, **cvc4+f-r**, and **cvc4+mace**. The second scatter plot clearly shows that the **cvc4+f** configuration generally requires less time and solves more benchmarks (723 vs. 664) than **cvc4+f-r**, confirming the usefulness of a region-based approach for clique detection. The third scatter plot compares **cvc4+f** against **cvc4+mace**. The latter configuration was able to solve only 617 benchmarks and generally performed poorly on benchmarks with larger model size. The median model size of the 123 benchmarks solved only by **cvc4+f** was 17, whereas the median size of the 13 benchmarks solved only by **cvc4+mace** was 10. This suggests that for larger cardinalities **cvc4+mace** suffers from the model symmetries created by the introduction of domain constants, something that **cvc4+f** avoids.

5.7.2 Finite Model Finder Evaluation

We provide results on CVC4 with finite model finding for three sets of benchmarks coming from different formal methods applications, including verification and automated theorem proving.

In this section, we will refer to various configurations of CVC4 based on the features they include. Configuration **cvc4+f** uses techniques for finding finite models. Additionally, configurations containing **m** in their suffix use the model-based quantifier instantiation heuristic described in Section 5.4.1, configurations with **M** use the model-based quantifier instantiation heuristic described in Section 5.4.2, configurations with **i** use heuristic instantiation as described in 5.6.1, configurations with **s** add ground clauses for breaking symmetry based on sort inference described in Section 5.6.2, and configurations with **S** convert the problem into the inferred signature also described in Section 5.6.2. All configurations of CVC4 with finite model finding use techniques for relevancy as described in Section 5.6.3. In these experiments, configurations of **m** used fragmented models, and **M** used simple models.

In the implementation, for performance reasons, we disable various features that theoretically ensure finite model completeness and refutational completeness. First, we did not implement a fair strategy for multiple sorts, as mentioned in Section 5.2.4, since this source of non-termination was not observed in our experiments. Second, we do not constrain the selection of representative terms as mentioned in Section 5.3.1, since we found that allowing the ground solver to choose representative terms leads to simpler ground conflicts that are found more quickly. Third, we do not eliminate nested quantifiers of negative polarity as mentioned in Appendix A.4, since introducing Skolem symbols leads to additional work for the model construction procedure, namely, it must find an explicit model for symbols not in the original problem. The performance degradation for each of these features is not very significant.

Nevertheless, since they have a detectable negative impact on performance overall, they are disabled by default in CVC4.

Experiments from Section 5.7.2.1 were run on a Linux machine with an 8-core 2.60GHz Intel® Xeon® E5-2670 processor. All others were run on a Linux machine with an 8-core 3.20GHz Intel® Xeon® E5-1650 processor with 16GB of RAM.

5.7.2.1 Intel benchmarks

We evaluated the overall effectiveness of CVC4's finite model finder for quantified SMT formulas taken from verification conditions generated by DVF [31], a tool used at Intel for verifying properties of security protocols and design architectures, among other applications. Both unsatisfiable and satisfiable benchmarks were produced, the latter by manually removing necessary assumptions from verification conditions. All benchmarks contain quantifiers, although only over free sorts, and span a wide range of theories, including linear integer arithmetic, arrays, EUF, and inductive datatypes.

For comparison we looked at the SMT solvers CVC3[8] [5] (version 2.4.1), Yices [25] (version 1.0.32), and Z3 [19] (version 4.1). We did not consider traditional theorem provers and finite model finders because they do not have built-in support for the theories in our benchmark set. All these solvers use E-matching as a heuristic method for answering unsatisfiable in the presence of universally quantified formulas. Z3 additionally relies on model-based quantifier instantiation techniques to

[8] CVC3 is the predecessor of CVC4. The latter was developed from scratch, and does not have code in common with CVC3.

Sat	german (45)		refcount (6)		agree (42)		apg (19)		bmk (37)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
yices	2	0.0	0	0.0	0	0.0	0	0.0	0	0.0
z3	45	1.1	1	7.0	0	0.0	0	0.0	0	0.0
cvc4+i	2	0.0	0	0.0	0	0.0	0	0.0	0	0.0
cvc4+f	**45**	0.3	**6**	0.1	42	15.5	18	200.0	36	1201.5
cvc4+f-r	**45**	0.3	**6**	0.1	42	18.6	15	364.3	34	720.4
cvc4+fi	45	0.4	**6**	0.1	**42**	14.2	19	492.8	36	831.0
cvc4+fm	**45**	0.3	**6**	0.1	42	23.6	**19**	210.2	37	375.1
cvc4+fmi	**45**	0.3	**6**	0.1	42	16.4	19	221.1	**37**	176.8

Unsat	german (145)		refcount (40)		agree (488)		apg (304)		bmk (244)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	145	0.4	40	0.2	457	6.8	267	77.0	229	76.2
yices	145	1.8	40	7.0	488	1475.4	304	35.8	244	25.3
z3	145	1.9	40	0.9	**488**	10.6	304	12.2	244	5.3
cvc4+i	**145**	0.1	40	0.2	484	6.8	**304**	11.2	**244**	2.9
cvc4+f	145	0.8	40	0.4	476	3782.1	298	2252.5	242	1507.0
cvc4+f-r	145	0.4	40	0.2	475	1574.3	294	3836.0	240	1930.5
cvc4+fi	145	0.7	**40**	0.1	488	188.7	302	342.0	244	660.3
cvc4+fm	145	0.4	40	0.3	471	5018.2	300	1122.7	242	834.1
cvc4+fmi	145	0.3	**40**	0.1	488	185.9	302	339.8	244	668.5

Figure 5.7. Results for satisfiable and unsatisfiable Intel (DVF) benchmarks. All times are in seconds.

be able to detect satisfiable quantified problems in some cases [28].

The results, separated into unsatisfiable and satisfiable instances, are shown in Figure 5.7 for five classes of benchmarks and a timeout of 600 seconds per benchmark. The first two classes, **refcount** and **german**, represent verification conditions for systems described in [31]; benchmarks in the third are taken from [59]; the last two classes are verification problems internal to Intel. Due to proprietary restrictions on these benchmarks, we report results for an older version of CVC4 (version 1.0) that did not incorporate some of the previously mentioned enhancements.

For the satisfiable benchmarks, our finite model finder is the only tool capable of solving any instance in the last three benchmark classes. In fact, **cvc4+f** is able to solve all but two, and most of them in less than a second. When extended to

include techniques for model-based quantifier instantiation (configurations **cvc4+fm** and **cvc4+fmi**), we are able to solve all satisfiable benchmarks within the timeout. By comparing **cvc4+f** against **cvc4+f-r**, we see that the region-based approach for recognizing cliques is beneficial, particularly for the harder classes where the latter configuration solves fewer benchmarks within the timeout. The model sizes found for these benchmarks were relatively small; only a handful had a model with sort cardinalities larger than 4. To our knowledge, our model finder is the only tool capable of solving these benchmarks.

For the unsatisfiable benchmarks, Yices and Z3 can solve all of them, with Z3 being much faster in some cases. We observe that CVC4 with finite model finding is orders of magnitude slower than the SMT solvers on these benchmarks. This is, however, to be expected since it is geared towards finding models, and applies exhaustive instantiation with increasingly large cardinality bounds, which normally delays the discovery that the problem is unsatisfiable regardless of those bounds.

However, we found that each unsatisfiable problem can be solved by either **cvc4** or **cvc4+fmi**, and in less than 3s. Additionally, configuration **cvc4+fmi** solves all unsatisfiable benchmarks within 900s, suggesting that CVC4's model finder makes consistent progress towards answering unsatisfiable on provable DVF verification conditions. From the perspective of verification tools, the results here seem promising. A common strategy for handling a verification condition would be to first use an SMT solver hoping that it can quickly find it unsatisfiable with E-matching techniques; and then resort to finite model finding if needed to either answer unsatisfiable, or produce

a model representing a concrete counterexample for the verification condition.

5.7.2.2 TPTP benchmarks

We considered benchmarks from a recent version of the TPTP library [57] (5.4.0), a widely-used library from the automated theorem proving community. The benchmarks from this library contain no theory reasoning (other than equality), and are composed mostly of quantified formulas.

We compared CVC4 (version 1.2) against other SMT solvers including Z3 (version 4.3) and CVC3 (version 2.4.1), as well as various automated theorem provers and model finders for first order logic, including Paradox [16] and iProver [39] (version 0.99). Paradox is a MACE-style model finder that uses preprocessing optimizations such as sort inference and clause splitting, among others, and then encodes to SAT the original problem together with increasingly looser constraints on the size of the model. iProver is an automated theorem prover based in the Inst-Gen calculus that can also run in finite model finding mode (**iprover+f**). In that mode, it incrementally bounds model sizes in a manner similar to MACE-style model finding. However, it encodes the whole problem into the EPR fragment, for which it is a decision procedure. Since these two tools are limited to classical first-order logic with equality, we considered only the unsorted first-order benchmarks of TPTP.

Figure 5.8 shows results for benchmarks from the TPTP library that are known to be satisfiable or unsatisfiable. All experiments were run with a 10 second timeout per benchmark. The benchmarks were placed into (exactly one) category based on its logical and syntactic characteristics, where EPR includes benchmarks that reside

	Sat					Unsat				
	EPR (392)	**NEQ** (639)	**SEQ** (340)	**PEQ** (624)	**TOTAL** (1995)	**EPR** (1114)	**NEQ** (1594)	**SEQ** (7875)	**PEQ** (2003)	**TOTAL** (12586)
z3	320	155	164	249	888	**989**	**412**	**3310**	1320	**6031**
cvc3	27	0	0	0	27	787	381	3019	883	5070
iprover	**363**	128	107	396	994	835	105	2690	**1523**	5153
iprover+f	362	226	178	468	1234	213	1	121	48	383
paradox	340	**304**	185	**526**	**1355**	723	17	339	186	1265
cvc4+i	32	0	0	0	32	821	383	3152	1045	5401
cvc4+f	295	178	143	375	991	759	247	887	651	2544
cvc4+fm	298	221	178	391	1088	759	169	1010	703	2641
cvc4+fM	301	235	**200**	395	1131	759	198	1073	733	2763
cvc4+fMi	292	207	153	385	1037	762	236	1281	746	3025
cvc4+fMs	296	242	197	382	1117	765	199	1230	798	2992
cvc4+fMS	305	244	199	410	1158	771	201	1356	896	3224

Figure 5.8. Results for TPTP benchmarks. All experiments were run with a 10 second timeout.

in the effectively propositional fragment, NEQ are benchmarks that do not contain any equality reasoning, SEQ are benchmarks containing some equality, and PEQ are benchmarks containing only pure equality.

For satisfiable benchmarks, CVC4's model finder with exhaustive instantiation (**cvc4+f**) solves 991 benchmarks. Using model-based quantifier instantiation, that number goes up to 1088 with the algorithm from Section 5.4.1 (**cvc4+fm**), and up to 1131 with the algorithm from Section 5.4.2 (**cvc4+fM**). Using further optimizations for the latter of these algorithms, both performing heuristic instantiation (**cvc4+fMi**) and adding ground clauses for symmetry breaking (**cvc4+fMs**) led to finding fewer satisfiable benchmarks, while determining the satisfiability of the problem in the inferred signature based on sort inference (**cvc4+fMS**) solved the most satisfiable benchmarks of any configuration of CVC4, solving 1158 within the timeout.

While CVC4 solves more than Z3, which finds 888 satisfiable benchmarks, our model finder still trails the overall performance of the other model fingers on these

problems. Paradox was the overall best solver, finding 1355 satisfiable benchmarks. We attribute this to the fact that we have not implemented advanced preprocessing techniques, such as clause splitting, that have been shown to be critical for finding finite models of TPTP benchmarks. Nevertheless, CVC4's model finder is capable of solving benchmarks that neither Paradox nor iProver can solve. In particular, it solves more satisfiable benchmarks (200) than any other solver for classes of problems having some equality reasoning (SEQ). Collectively, some configuration of CVC4 with finite model finding was able to solve 41 satisfiable benchmarks that neither Paradox nor iProver was able to solve. Additionally, some configuration of CVC4 with finite model finding was able to solve 3 satisfiable benchmarks with 1.0 difficulty rating, which means that no known ATP system had solved these problems when version of 5.4.0 of the TPTP library was released (in June of 2012).

Figure 5.8 also shows results for unsatisfiable problems. Although these results are not comparable to those achieved by state-of-the-art theorem provers, such as Vampire and E, we note that Z3 solves the most benchmarks, 6031. Here, **cvc4+fMS** was the best configuration of CVC4 with finite model finding, solving 3224 within the timeout. While finite model finding configurations solved considerably fewer than using heuristic instantiation alone, some configuration of CVC4 with finite model finding solves 116 unsatisfiable benchmarks that were unable to be solved by any other solver in these experiments, including Z3.

To further evaluate the impact of model-based quantifier instantiation on our model finder, we recorded statistics on the domain size of quantified formulas in

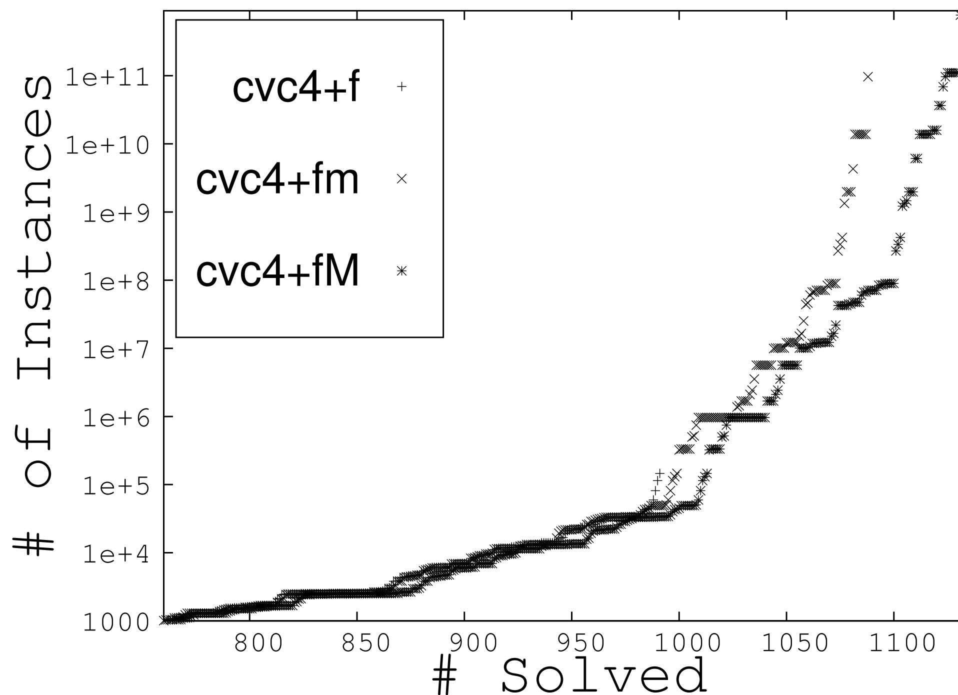

Figure 5.9. Satisfiable TPTP problems with and without model-based instantiation. A point (x, y) on this graph says the configuration solves x benchmarks each having at most y ground instances of quantified formulas.

benchmarks solved by its various configurations. We measured the total number of possible ground instances for all quantified formulas in the smallest model for that benchmark (a quantified formula over n variables each with domain size k has k^n instances). For a problem with d total instances, the configuration **cvc4+f** must explicitly generate these d instances, while a model-based configuration may avoid doing so.

For these experiments, **cvc4+f** was only able to solve 2 problems having more than 100K instances, the maximum having around 146K instances. On the other hand, **cvc4+fm** was capable of solving 92 problems having more than 100K in-

stances, with the largest having more than 96.8 billion instances. Techniques from Section 5.4.2 (configuration **cvc4+fM**) led to even better performance, solving 121 problems having more than 100k instances, with the largest having more than 775 billion instances. This information is plotted in Figure 5.9, showing how the model-based instantiation approaches improves the scalability of our model finder, and allows it to solve benchmarks where exhaustive instantiation is clearly infeasible. Note that model finders such as Paradox have other ways of handling the explosion in the number of instances, namely by minimizing the number of variables per clause. Coupling these techniques with model-based techniques could lead to additional improvements in scalability. Since techniques for reducing variables in clauses rely on introducing new symbols into the problem, we have found that they have a negative impact on performance for several classes of benchmarks, and thus are disabled by default in CVC4.

Recently, CVC4 participated in CASC 24, a competition evaluating the performance of automated theorem provers on selected TPTP benchmarks. In the first-order non-theorems division (FNT), CVC4 finished 3rd out of 8 solvers, solving 96 of 150 problems, just behind Paradox, which solved 99. The newest version of iProver won the competition solving 122 problems. At the time of the competition (June of 2013), CVC4 did not incorporate enhancements based on sort inference.

5.7.2.3 Isabelle benchmarks

Recent work has shown that SMT solvers are effective at discharging proof obligations for Isabelle, a generic proof assistant [52]. The performance of these

solvers can benefit from an encoding that makes use of theories [8]. We considered a set of 13,041 benchmarks corresponding to both provable and unprovable proof goals, corresponding to a superset of those discussed in [8]. Most benchmarks in this set contain quantifiers, and a significant portion contain integer arithmetic. For many of them, the quantification is limited to the free sorts, thus making our finite model finding approach applicable. Since CVC4 does not yet have support for non-linear arithmetic, we report results only for the 11,130 benchmarks that do not contain (non-trivial) non-linear arithmetic constraints.

These benchmarks contained annotations corresponding to user-provided patterns and weight values that were added to optimize the performance of Z3. We report on Z3 with patterns (configuration **z3+p**) and without patterns (configuration **z3**). We omit results for CVC3 and CVC4 with patterns, which both performed slightly worse when patterns were provided.

The results are shown in Figure 5.10. For satisfiable benchmarks, all configurations of CVC4's model finder find more satisfiable problems than Z3, which finds only 177 of them overall when using patterns. The model-based quantifier instantiation technique from Section 5.4.1 (configuration **cvc4+fm**) was slightly less effective than naive instantiation (configuration **cvc4+f**) which solves 738, suggesting that useful instantiations were missed during the instance selection procedure. On the other hand, the techniques from Section 5.4.2 were more effective than both of these configurations, as **cvc4+fM** solved 748 overall. Sort inference techniques (configurations **cvc4+fMs** and **cvc4+fMS**) were able to infer many subsorts for these benchmarks,

Sat	**Arrow**	**FFT**	**FTA**	**Hoare**	**NSS**	**QEpres**	**SNorm**	**TSq**	**TSafe**	**TOTAL**
cvc3	0	9	0	0	0	0	0	8	0	17
z3	2	19	24	46	10	38	1	15	12	167
z3+p	1	19	24	46	10	47	1	17	12	177
cvc4+i	0	9	0	0	0	0	0	8	0	17
cvc4+f	26	123	**163**	149	56	75	12	**50**	84	738
cvc4+fi	26	133	158	**155**	**61**	80	12	44	**87**	756
cvc4+fm	22	120	152	147	36	75	12	46	87	697
cvc4+fM	28	126	163	151	44	94	12	43	87	748
cvc4+fMi	**31**	**136**	161	154	61	**101**	**12**	44	85	**785**
cvc4+fMs	28	125	156	148	44	93	12	43	87	736
cvc4+fMS	28	127	162	150	46	95	12	43	87	750

Unsat	**Arrow**	**FFT**	**FTA**	**Hoare**	**NSS**	**QEpres**	**SNorm**	**TSq**	**TSafe**	**TOTAL**
cvc3	**287**	**250**	**877**	**577**	102	**291**	206	**552**	216	**3358**
z3	166	238	733	497	105	251	**240**	495	297	3022
z3+p	254	230	797	507	**135**	242	240	491	**329**	3225
cvc4+i	253	233	749	476	99	265	234	523	267	3099
cvc4+f	123	94	350	209	41	99	83	361	127	1487
cvc4+fi	155	164	509	374	37	168	100	452	195	2154
cvc4+fm	112	86	357	212	26	119	82	349	120	1463
cvc4+fM	88	92	381	202	29	109	93	365	149	1508
cvc4+fMi	154	164	515	371	37	167	100	452	195	2155
cvc4+fMs	76	89	388	211	33	111	91	365	164	1528
cvc4+fMS	88	94	386	225	29	111	93	365	173	1564

Figure 5.10. Results for satisfiable and unsatisfiable Isabelle benchmarks. Tables show the number of problems solved for various classes within a 10 second timeout.

and led to a slight improvement for the latter configuration, solving 750. Using heuristic E-matching noticeably improved the search for models, as configuration **cvc4+fi** solves 756 satisfiable benchmarks. Using both model-based instantiation and heuristic instantiation, configuration **cvc4+fMi**, found more satisfiable problems (785) than any other configuration.

For unsatisfiable problems, cvc3 is the overall winner, solving 3,358, which was more than both z3 with patterns and **cvc4+i** which solved 3,225 and 3,099 respectively. Configurations of cvc4 with finite model finding generally solves less unsatisfiable benchmarks, but is orthogonal to other solvers and configurations. In these experiments, 170 unsatisfiable benchmarks that cvc3 cannot solve are solved by at least one configuration of cvc4 with finite model finding. Similarly, a configuration

of CVC4 with finite model finding solves 365 unsatisfiable benchmarks that Z3 cannot, and 229 that **cvc4+i** cannot.

We are investigating new ways where model finding can be used in the context of automated theorem proving systems. Some of these systems are based on selecting a set of relevant background axioms that may be sufficient for proving a conjecture. In this case, a model finder could be used to identify queries where searching for a proof is guaranteed to fail.

CHAPTER 6

EXTENSIONS TO OTHER DOMAINS

In this chapter, we mention further domains for which a finite model finding approach can be applied, including integer quantification when finite bounds can be inferred and model finding for the theory of strings.

6.1 Bounded Integer Quantification

In this section, we examine how an approach for finite model finding can be extended to handle problems with quantification over the built-in integer sort Int. Our approach will be limited to quantified formulas where bounds can be inferred for all quantified integer variables.

Definition 13 *A quantified formula has bounded integer quantification if and only if it is equivalent to* $\forall\, x_1, \ldots x_n : \mathsf{Int}.\ \ell_1 \leq x_1 \leq u_1 \wedge \ldots \wedge \ell_n \leq x_n \leq u_n \Rightarrow \psi$, *where* (i) $n > 0$, *and* (ii) $x_j \notin FV(l_i, u_i)$ *for* $1 \leq i \leq j \leq n$.

We assume we are using the model finding approach outlined in Section 5.1. Analogous to the methods described in the previous chapter, after finding terms that effectively bind the range of instances of a quantified formula φ we need to consider, we will minimize the values of these terms in a candidate model using a strategy in DPLL($T_1, \ldots, T_m$), construct a candidate model $\mathcal{M}$ from satisfying assignments, and use this model to guide how we instantiate the quantified formula φ.

```
proc infer_bounds(φ, X) ≡
  B := ∅;
  while B ≠ X
    for x ∈ X, x ∉ B
      if φ ⊨ ℓ ≤ x ≤ u, FV(ℓ, u) ⊆ B
        B := B ∪ {x}
        lower(x) := ℓ, upper(x) := u
      end
    end
  end
```

Figure 6.1. The infer_bounds procedure. Finds terms $\mathsf{lower}(x)$ and $\mathsf{upper}(x)$ for each $x \in X$, where X is the set of free integer variables in φ.

6.1.1 Inferring Bounds

To begin, we must determine if each quantified formula $\forall \mathbf{x}.\varphi$ in our input has bounded integer quantification. For each integer variable $x \in \mathbf{x}$, we wish to find terms corresponding to the lower and upper bound for x, which we will denote as $\mathsf{lower}(x)$ and $\mathsf{upper}(x)$. Figure 6.1 gives a method infer_bounds for finding these terms. For determining cases where $\varphi \models \ell \leq x \leq u$, we consider literals equivalent to $c \cdot x + d \cdot \mathbf{y} \leq 0$ having polarity p in φ, and where $c \neq 0$. From this, a term of the form $-\frac{d}{c} \cdot \mathbf{y}(\pm 1)$ is either a candidate lower or candidate upper bound for x, depending on the sign of c and the polarity of the literal p. On each iteration, we infer bounds on variables for which both an upper and lower bound (whose free variables are in B) can be found.

Consider the term $\mathsf{upper}(x) - \mathsf{lower}(x)$ for a bounded integer variable x in our problem. If this term is ground, our strategy for DPLL($T_1, \ldots, T_m$) will attempt to minimize its value in the current assignment M. If this term contains free variables $\mathbf{y}$,

we will ensure that $(\mathsf{upper}(x) - \mathsf{lower}(x))\{\mathbf{y} \mapsto \mathbf{v}\} \leq k$ is satisfied for some values $\mathbf{v}$, where k is a fresh constant. We write $\mathsf{range}(x)$ to denote the term $\mathsf{upper}(x) - \mathsf{lower}(x)$ if this term is ground, or the fresh constant k introduced for x otherwise.

6.1.2 Establishing Finite Bounds

Our approach for bounded integer quantification relies on a strategy for DPLL($T_1, \ldots, T_m$) which establishes finite bounds for each quantified variable. Following the conventions from Section 4.2.1, at weak effort, we check if there exists any bounded integer variable x such that no literal equivalent to $\mathsf{range}(x) < r$ exists in M. If so, let r' be the smallest integer such that $M \models \mathsf{range}(x) \geq r'$, or 0 otherwise. Such a lower bound on $\mathsf{range}(x)$ can typically be obtained from the Simplex procedure used by most SMT solvers. Using the rule $\textbf{Learn}_i$, we add the lemma $(\mathsf{range}(x) < r' \vee \neg\mathsf{range}(x) < r')$. Then, if there exists any bounded integer variable x such that the literal equivalent to $\mathsf{range}(x) < r$ does not exist in M, we may choose $\mathsf{range}(x) < r$ with positive polarity as the literal for which **Decide** is applied.

6.1.3 Constructing Candidate Models

Once a satisfying assignment M is found for our set of clauses F, we construct a candidate model $\mathcal{M}$ satisfying M. In the remainder of this section, we assume we are given an evaluation map $\mathcal{A}_M$ for M. For now, we address how definitions for functions containing only integer arguments are constructed.

6.1.3.1 Representing Function Definitions

We will use the representation for functions as provided in Section 5.3.2, where our abstract values are extended to incorporate *intervals*:

$$u := \bot \mid v \mid * \mid [v_1, v_2] \tag{6.1}$$

Here, $[v_1, v_2]$ represents any integer value between v_1 and v_2, where v_i is either an integer value, $-\infty$, or ∞. We assume similar notions from Section 5.3.2 for abstract values in this extension. For an interval $[v_1, v_2]$, we define its concretization $\gamma([v_1, v_2])[x]$ as the formula $v_1 \leq x \leq v_2$. It can be shown that the meet of two abstract values remains well-defined in this extension, i.e. $[v_1, v_2] \bigtriangleup [w_1, w_2] = [max(v_1, w_1), min(v_2, w_2)]$. Note that if $v_1 > v_2$, then $[v_1, v_2]$ is equivalent to $\bot$. In the following, we may write $*$ as shorthand for $[-\infty, \infty]$.

6.1.3.2 Constructing Function Definitions

Similar to Section 5.3, we consider only candidate models $\mathcal{M}$ that are induced by Σ-maps. Thus it suffices to show how definitions D_f are constructed for each function f, given a satisfying assignment M.

Say we are given a function f of sort $\mathsf{Int} \times \ldots \times \mathsf{Int} \to S$ for some sort S. As before, our construction of D_f will be based on terms in $\mathbf{T}_M$ with top symbol f. Given our evaluation map $\mathcal{A}_M$, let $\mathsf{terms}_i(f(t_1, \ldots, t_n)) \subseteq \mathbf{T}_M$ be the set $\{f(s_1, \ldots, s_n) \mid \forall 1 \leq j < i. \mathcal{A}_M(t_j) = \mathcal{A}_M(s_j)\}$. Let $\mathsf{ival}_i(f(t_1, \ldots, t_n))$ be the interval whose lower bound is $-\infty$ if $\mathcal{A}_M(t_i) = min(\{\mathcal{A}_M(s_i) \mid f(s_1, \ldots, s_n) \in \mathsf{terms}_i(f(t_1, \ldots, t_n))\})$ and $\mathcal{A}_M(t_i)$ otherwise, and whose upper bound is equal to $min(\infty, \{\mathcal{A}_M(s_i) - 1 \mid f(s_1, \ldots, s_n) \in \mathsf{terms}_i(f(t_1, \ldots, t_n)), \mathcal{A}_M(s_i) > \mathcal{A}_M(t_i)\})$. For example, if $f(1, 3)$,

$f(1,5)$, and $f(2,2)$ are in $\mathbf{T}_M$, then $\mathsf{terms}_2(f(1,3)) = \{f(1,3), f(1,5)\}$, and $\mathsf{ival}_2(f(1,3)) = [-\infty, 4]$.

We construct the definition D_f for function symbols $f \in \Sigma$, where f is of sort $\mathsf{Int} \times \ldots \times \mathsf{Int} \rightarrow S$ in the following way.

Construction of D_f: If $\mathbf{T}_M$ contains at least one term of the form $f(t_1, \ldots, t_n)$, then D_f is a definition consisting of:

$$(\mathsf{ival}_1(t), \ldots, \mathsf{ival}_n(t)) \rightarrow \mathcal{A}_M(t) \in D_f \qquad \text{for each } t \text{ of the form } f(t_1, \ldots, t_n) \in \mathbf{T}_M$$

Otherwise, D_f is $(*, \ldots, *) \rightarrow v$ for some value v.

It is immediate that all pairs of entries $\mathbf{c} \rightarrow v$ and $\mathbf{d} \rightarrow w$ are such that $\mathbf{c}$ and $\mathbf{d}$ are incompatible. Thus, it is easy to show that $\mathcal{M}$ is consistent with our evaluation map $\mathcal{A}_M$ and thus satisfies M.

Example 29 *Say our evaluation map $\mathcal{A}_M$ is $\{f(1,3) \mapsto 0, f(1,5) \mapsto 3, f(2,1) \mapsto 5\}$. Then, D_f is $([-\infty,1],[-\infty,4]) \rightarrow 0,\ ([-\infty,1],[5,\infty]) \rightarrow 3,\ ([2,\infty],[-\infty,\infty]) \rightarrow 5$.*

Although not shown here, if a function f contains arguments that are both integer and uninterpreted sorts, we may combine the aforementioned model construction for D_f with the one mentioned in Section 5.3.3. The idea is to partition terms in $\mathbf{T}_M$ with top symbol f into sets whose non-integer arguments evaluate to the same tuple of values in $\mathcal{A}_M$ before applying the model construction procedure described in this section.

```
proc bound_int_qi(M, φ, i, σ, e_ℓ, e_u) ≡
  if i > n
    return {σ}
  else if Mσ⟦u_i − ℓ_i⟧ > M⟦range(x_i)⟧
    apply Learn_i to ((u_i − ℓ_i)σ ≤ range(x_i))
    return fail
  else
    S := ∅
    for j = 0 ... (min(Mσ⟦u_i⟧, e_u.i) − max(Mσ⟦ℓ_i⟧, e_ℓ.i))
      S := S ∪ bound_int_qi(M, φ, i + 1, σ ∪ {x_i ↦ ℓ_iσ + j}, e_ℓ, e_u)
    end
    return S
  end
```

Figure 6.2. The bound_int_qi procedure. Given a candidate model $\mathcal{M}$ and $\varphi = \forall\, x_1, \ldots x_n : \mathsf{Int}.\; \ell_1 \leq x_1 \leq u_1 \wedge \ldots \wedge \ell_n \leq x_n \leq u_n \Rightarrow \psi$, this procedure, when successful, returns all relevant substitutions for φ whose values are between bounds $\mathbf{e}_\ell$ and $\mathbf{e_u}$. The procedure is called initially with $i = 1$ and $\sigma = \emptyset$.

6.1.4 Quantifier Instantiation

Similar to the methods from the previous chapter, after we construct a candidate model $\mathcal{M}$ from our satisfying assignment M, we apply a quantifier instantiation heuristic to choose a set of substitutions $I_\mathbf{x}$ for each quantified formula φ that is active in M. If φ has bounded integer quantification, a naive approach would be to consider the full range of instantiations of φ without considering which instantiations are already true in the model. To do this, we apply the method shown in Figure 6.2. with $\mathbf{e}_\ell = (-\infty, \ldots, -\infty)$ and $\mathbf{e_u} = (\infty, \ldots, \infty)$.

Consider a quantifier $\forall \mathbf{x}.\varphi$ having bounded integer quantification, where φ is equivalent to $\ell_1 \leq x_1 \leq u_1 \wedge \ldots \wedge \ell_n \leq x_n \leq u_n \Rightarrow \psi$. As before, to improve scalability, in some cases we may use a model-based approach for quantifier instantiation. In particular, if ψ is model-checkable, we compute $D_{\lambda \mathbf{x}.\psi}$ using the methods

from Section 5.4.2.2. Then, for each $([\ell'_1, u'_1], \dots, [\ell'_n, u'_n]) \to \textbf{false} \in D_{\lambda\mathbf{x}.\psi}$, we add one substitution to $I_{\mathbf{x}}$ for $\forall\mathbf{x}.\varphi$. Because of our model construction, which constructs definitions that are comprised of disjoint interval entries, it can be shown that $[\![\gamma(D_{\lambda\mathbf{x}.\psi})[v_1, \dots, v_n]]\!] = \textbf{false}$ for all $v_1, \dots v_n$ where $\ell'_1 \leq v_1 < u'_1 \dots \ell'_n \leq v_n < u_n$. We call the bound_int_qi procedure with with $\mathbf{e}_\ell = (\ell'_1, \dots, \ell'_n)$ and $\mathbf{e}_\mathbf{u} = (u'_1, \dots, u'_n)$, and add the first substitution it returns to $I_{\mathbf{x}}$ for $\forall\mathbf{x}.\varphi$.

Example 30 *Say we wish to determine the satisfiability of the set of clauses* $\{f(3) \approx 1, f(90) \approx -1, \varphi\}$*, where* φ *is* $\forall x.5 \leq x \leq 70 \Rightarrow f(x) \approx 0$*. After finding a satisfying assignment* $f(3) \approx 1, f(90) \approx -1$*. We construct a candidate model induced by the definition:*

$$D_f = ([-\infty, 89]) \to 1, ([90, \infty]) \to -1$$

When performing model-based quantifier instantiation on φ*, we calculate* $D_{\lambda x.f(x)\approx 0}$*, obtaining* $([-\infty, \infty]) \to \textbf{false}$*. Our quantifier instantiation heuristic adds the instance* $\varphi[5/x]$ *to our set of clauses. After finding a satisfying assignment for our new set of clauses, we construct a candidate model induced by the definition:*

$$D_f = ([-\infty, 4]) \to 1, ([5, 89]) \to 0, ([90, \infty]) \to -1$$

We will calculate $D_{\lambda x.f(x)\approx 0}$ *again for this candidate model, obtaining* $([-\infty, 4]) \to$ $\textbf{false}$*,* $([5, 89]) \to \textbf{true}$*,* $([90, \infty]) \to \textbf{false}$*. After processing the first and third entries, our quantifier instantiation algorithm will terminate with no instances produced, and we will answer satisfiable.* ■

Example 31 *Say we wish to determine the satisfiability of the set of clauses* $\{\neg P(0,0),$ $\forall x_1 x_2.(0 \leq x_1 \leq 5 \wedge 0 \leq x_2 \leq f(x_1)) \Rightarrow P(x_1, x_2)\}$. *As mentioned, to bound the values of variable* x_2*, we introduce a constant* k *such that we will require* $f(x) \leq k$ *for all* x*. We add the lemma* $(k < 0 \vee \neg k < 0)$*, and decide on* $k < 0$*. After finding a satisfying assignment, we construct a candidate model* $\mathcal{M}$ *induced by the definitions:*

$$D_P = (*,*) \to \textbf{false}, D_f = (*) \to 0, D_k = () \to -1$$

We compute $D_{\lambda x_1\, x_2.P(x_1,x_2)}$*, obtaining* $(*,*) \to \textbf{false}$*. While applying* bound_int_qi *for our quantified formula, we find that* $\mathcal{M}[\![f(0)]\!] = 0 > -1 = \mathcal{M}[\![k]\!] = \mathcal{M}[\![\mathsf{range}(x_2)]\!]$*. Thus, our quantifier instantiation heuristic fails, and instead we add the lemma* $f(0) \leq k$*. After finding another satisfying assignment, we build a new candidate model where* $D_f = (*) \to -1$*, our quantifier instantiation heuristic terminates with no instances, and we answer satisfiable.* ■

6.1.5 Properties

Here, we show that our approach for handling bounded integer quantification is sound, and give a sketch of how it can be extended to be *model complete*. In other words, given an input F_0 where all quantified formulas in F_0 have bounded integer quantification, if F_0 is satisfiable, then our approach will terminate with a model.

To show soundness, we must show that all lemmas added by our procedure preserve the satisfiability of F_0. Applications of quantifier instantiation and splitting lemmas clearly do so. The only other lemmas we add are of the form $((u_i - \ell_i)\sigma \leq \mathsf{range}(x))$ during the bound_int_qi procedure. Due to our definition of $\mathsf{range}(x)$, in this case $(u_i - \ell_i)$ must be a non-ground term and $\mathsf{range}(x)$ must be a fresh constant.

Since $\mathsf{range}(x)$ is a fresh constant, we may add an arbitrarily large (finite) number of clauses of this form without affecting the satisfiability of F_0.

To show model completeness, we extend our approach in the following way. Let B be the set of all (bounded integer) variables occurring in F_0. We incorporate a fair strategy when searching for models for F_0 in a manner similar to fixed-cardinality DPLL($T_1, \ldots, T_m$) for multiple sorts. Assuming we use splitting on demand for introducing necessary literals, instead of deciding on literals of the form $\mathsf{range}(x) < r$ for each $x \in B$, we decide on a literals of the form $\Sigma_{x \in B} \mathsf{max}(\mathsf{range}(x), -1) < r$, where $\mathsf{max}(s, t)$ is shorthand for the term $ite(s < t, t, s)$.

Now, say that F_0 is satisfiable with model $\mathcal{M}$. For each $x \in B$, let $S(x)$ be the (finite) set of substitutions with domain $FV(\mathsf{upper}(x) - \mathsf{lower}(x))$ that map each variable y to a term from the set $\{\mathsf{lower}(y)\sigma + i \mid \sigma \in S(y), 0 \leq i \leq \mathcal{M}\sigma[\![\mathsf{upper}(y) - \mathsf{lower}(y)]\!]\}$. It can be shown that our procedure will terminate (at worst) when considering the bound $r = \Sigma_{x \in B} max(\{\mathcal{M}\sigma[\![\mathsf{upper}(x) - \mathsf{lower}(x)]\!] \mid \sigma \in S(x)\})$ for the term $\Sigma_{x \in B} \mathsf{max}(\mathsf{range}(x), -1)$. Now, let $S'(x)$ be the (finite) set of substitutions with domain $FV(\mathsf{upper}(x) - \mathsf{lower}(x))$ that map each variable y to a term from the set $T(y) = \{\mathsf{lower}(y)\sigma + i \mid \sigma \in S'(y), 0 \leq i \leq r + \mid B \mid\}$, which we similarly call $T(x)$ for any variable x. Note that $T(x)$ is finite for all variables $x \in B$. For a quantified formula φ with bounded integer quantification, our procedure will only add instantiations of the form $\varphi\sigma$ where each x in the domain of σ is mapped to a term from $T(x)$. Since the number of instantiations of this form is finite, our procedure will only perform a finite number of instantiation rounds before terminating with a model.

	sat (263)		unsat (843)	
	solved	time	solved	time
z3	257	957.9	843	20.3
cvc4+i	0	0.0	**843**	17.4
cvc4+fi	**263**	90.8	843	308.7

Figure 6.3. Results for Intel benchmarks containing bounded integer quantification. All times are in seconds.

6.1.6 Results

We evaluated the performance of our approach for bounded integer quantification on a set of benchmarks created by the Intel Corporation corresponding to properties of memory within bounded integer ranges. Satisfiable benchmarks were generated by removing necessary assumptions from proof goals. The benchmarks contain ground theory constraints over arrays, datatypes, uninterpreted functions and integers. All non-trivial quantified formulas in these benchmarks were either in the bounded integer fragment described above, or otherwise had quantification only over uninterpreted sorts. [1] All bounds on integer variables in these problems (terms $\ell_1, \ldots, \ell_n, u_i \ldots, u_n$ from Defintion 13) were symbolic constants.

The results are shown in Figure 6.3 for unsatisfiable and satisfiable benchmarks. We ran the SMT solver z3, CVC4 with heuristic instantiation (the configuration **cvc4+i**), and CVC4 with the methods described in this section as well as using heuristic instantiation (**cvc4+fi**). We did not use model-based quantifier instantiation for these experiments. All configurations were run with a 600 second timeout.

In our experiments, both Z3 and **cvc4+i** quickly solved all of the unsatisfiable

[1] A few unsatisfiable benchmarks had quantification over integers for the initialization of constant arrays.

benchmarks. Additionally, the configuration **cvc4+fi** solved all unsatisfiable benchmarks in less than 60 seconds. For satisfiable benchmarks, Z3 solved most of them, solving 257 out of 263 within the timeout. It was able to do so because the benchmarks were in the almost uninterpreted fragment, as described in [28]. Techniques for bounded integer quantification were effective for satisfiable benchmarks in this set, and had the best performance overall. The configuration **cvc4+fi** solved each of the 263 satisfiable benchmarks in less than 20 seconds.

For most of these benchmarks, the bounded range of most quantified variables was relatively small. The range of most quantified formulas was around 2 to 4, while the largest range of bounded integer variable encountered in this set was 10. In other words, for a quantified formula $\forall x. \ell \leq x \leq u \Rightarrow P(x)$ where ℓ and u are (symbolic) ground integer constants, a model $\mathcal{M}$ was found where $\mathcal{M}[\![u - \ell]\!] = 10$. Other benchmarks had quantifiers where a model could be constructed such that all variables had bounded ranges that were negative, thus eliminating the need for quantifier instantiation altogether. Given these details, this means CVC4 with finite model finding was able to find counterexamples to the verification conditions from this set involving a relatively small number of memory addresses.

6.2 Strings

A finite model finding approach can be used for the theory of strings with length constraints. Consider a basic definition for this theory, where terms are either variables x, constants consisting of a sequence of characters from a finite alphabet, or concatenation of strings. We assume the signature of theory of strings also contains

a unary function for length, which we will write as $|\cdot|$, having type $\mathsf{String} \rightarrow \mathsf{Int}$. Various approaches having been proposed for handling string constraints in SMT, including the HAMPI solver [35], which solves constraints for fixed-size string and context-free language constraints by a reduction to bit-vector constraints.

Following a similar approach as previous sections, a finite model finding approach can be integrated into DPLL($T_1, \ldots, T_m$) for a set of clauses F, where T_i is the theory of strings with length constraints, in the following manner. Let $x_1, \ldots, x_n$ be the string variables occurring in F. When a weak effort check produces no conflicts, if there exists no literals of the form $| x_1 | + \ldots + | x_n | \leq k$ in our current assignment M, we find an (integer) lower bound k on the value of $| x_1 | + \ldots + | x_n |$ in M, where $k = 0$ if none exists. If necessary, we apply $\mathbf{Learn}_i$ to split on the literal $|x_1| + \ldots + |x_n| \leq k$ and subsequently decide (positively) on $|x_1| + \ldots + |x_n| \leq k$.

Assume we are given a decision procedure for strings of bounded length. If F has a model where $x_1, \ldots, x_n$ are interpreted as strings of finite length, the DPLL($T_1, \ldots, T_m$) procedure will terminate, noting that only a finite number of literals of the form $|x_1| + \ldots + |x_n| \leq k$ will be introduced as a result of this extension. We are currently working on an implementation of this approach in CVC4, as well as a theory solver for this theory.

CHAPTER 7

CONCLUSION

We developed a procedure for finite model finding in SMT that is efficient for many classes of problems that are of practical interest to formal methods applications. Experimental evidence shows that an implementation of these methods in the SMT solver CVC4 is a efficient approach for solving many classes of benchmarks, including verification conditions from industry, and benchmarks from automated theorem proving libraries. The implementation is highly competitive both with respect to other SMT solvers and automated theorem provers.

An efficient approach for finite model finding in SMT was made possible by a search strategy that establishes finite cardinality constraints, and a specialized ground theory solver for determining the satisfiability of these constraints. A key feature of this theory solver is that it performs an incomplete check for eagerly recognizing cardinality conflicts, allowing the solver to avoid parts of the search that are obviously infeasible.

For determining the satisfiability of quantified formulas, we introduced new methods for representing and constructing candidate models. Our representation of candidate models can be extended in various ways, including to intervals for functions with integer arguments. We believe the representation can be extended to incorporate models for other theories, including numeric intervals for bit-vectors and pattern matching constraints for inductive datatypes. We believe our model construction

also can be easily integrated with new frameworks for SMT, namely, those making model assignments explicit in the search [21].

We also introduced new algorithms for model-based quantifier instantiation to test whether candidate models satisfy universally quantified formulas, and for choosing relevant instances to refine the candidate model. The first algorithm relied on generalizing ground evaluations of quantified formulas, and grouping sets of instantiations that evaluate to the same value. The second of these algorithms was based on computing a representation of the interpretation of non-ground terms in the candidate model, thus allowing us to efficiently check when no instance of a quantified formula is falsified.

We introduced various enhancements for finite model finding in SMT, including heuristic instantiation, sort inference, and the use of relevancy for minimizing satisfying assignments. These techniques rule out spurious candidate models, reduce the amount of symmetry in the problem, and lower the overhead of checking candidate models. As demonstrated in our experiments, they improve the performance of the solver for both satisfiable and unsatisfiable benchmarks.

Finite model finding techniques also can be generalized to other domains of SMT, including bounded integer quantification and the theory of strings. In particular, we showed that an approach that is sound and model complete for problems where all quantified formulas have bounded integer quantification. Preliminary results show that these techniques are highly effective at solving problems of interest to verification and security applications.

Several important properties were shown for our approach for finite model finding in the presence of quantified formulas. Firstly, our procedure is finite model complete for any input where quantification is limited to uninterpreted sorts. This result was guaranteed by having a fair search strategy when multiple sorts are present, and by instantiating quantified formulas only with terms maximum depth. Secondly, our procedure is refutationally complete when using naive quantifier instantiation in the absence of background theories. This result was guaranteed by showing that the procedure fairly enumerates a finite set of ground instances that is sufficient for showing the input to be unsatisfiable. We conjecture this result can be generalized to some cases where background theories are also present at the ground level, but this is left for future work.

We believe the methods described in this thesis provide a starting point for most applications that rely on models for first-order quantified formulas in SMT. In many applications, an encoding can be used where quantification is limited to uninterpreted sorts by abstracting the domain of each quantified formula. Provided that a model in the resultant encoding implies that a model of interest exists in the original problem, the procedures mentioned in this thesis are applicable, and moreover are guaranteed to terminate successfully when a finite model exists.

APPENDIX A

PREPROCESSING

In this section, we describe preprocessing techniques applied to an input φ. When considering a combination of theories $T_1 \cup \ldots \cup T_n$ with signatures $\Sigma_1, \ldots, \Sigma_n$, we first perform a purification step so that each ground atomic subformula ψ of φ is over a single signature Σ_i for some i, or in other words, ψ is *pure*. If φ contains a ground atomic formula ψ that is not pure, we replace a pure subterm t of ψ of sort S with a constant c from a set of constants $\mathcal{C}_S$ shared by the signatures of each theory, and add the (pure) equality $t \approx c$ to our input. We repeat this process until all ground atomic formulas in our input are pure.

Let us now turn our attention to quantified formulas occurring in φ. We first rewrite all existential quantifiers in φ to universal quantifiers, rewriting $\exists \mathbf{x}\, \varphi$ to $\neg \forall \mathbf{x}\, \neg \varphi$. We then apply techniques described in A.1- A.4 to each $\forall \mathbf{x}\, \neg \varphi$ occurring in the resulting formula. Then, we convert φ to a set of clauses as described in A.5.

A.1 Negation Normal Form

We first rewrite the body of a quantified formula $\forall \mathbf{x}.\varphi$ such that the only logical connectives in φ are $\neg$, $\wedge$ and $\vee$, using the following rewrites.

$$\varphi_1 \Leftrightarrow \varphi_2 \rightarrow (\varphi_1 \wedge \varphi_2) \vee (\neg \varphi_1 \wedge \neg \varphi_2) \tag{A.1}$$

$$\varphi_1 \Rightarrow \varphi_2 \rightarrow \neg \varphi_1 \vee \varphi_2 \tag{A.2}$$

$$ite(\varphi_1, \varphi_2, \varphi_3) \rightarrow (\varphi_1 \wedge \varphi_2) \vee (\neg \varphi_1 \wedge \varphi_3) \tag{A.3}$$

Then, we assume a standard conversion to negation normal form, where every negation is applied directly to an atomic formula.

A.2 Miniscoping

We may apply miniscoping in two ways, as shown below.

$$\forall \mathbf{x}.(\varphi \vee \psi) \rightarrow (\forall \mathbf{x}.\varphi) \vee \psi \qquad \text{if } \mathbf{x} \cap FV(\psi) = \emptyset \tag{A.4}$$

$$\forall \mathbf{x}.(\varphi \wedge \psi) \rightarrow (\forall \mathbf{x}.\varphi) \wedge (\forall \mathbf{x}.\psi) \tag{A.5}$$

A.3 Destructive Equality Resolution

We may apply destructive equality resolution, which eliminates a variable from a quantified formula.

$$\forall x\mathbf{y}.(x \not\approx t \vee \varphi) \rightarrow \forall \mathbf{y}.(\varphi[t/x]) \qquad \text{if } x \notin FV(t) \tag{A.6}$$

A.4 Eliminating Nested Quantifiers

We may eliminated nested quantification from $\forall \mathbf{x}.\varphi$ in the following way. We say that a literal has positive (respectively negative) polarity in a formula if it occurs beneath an even (respectively odd) number of negations, and does not occur beneath any quantifier.

If there exists a subformula $\forall \mathbf{y}.\psi$ of φ that has positive polarity in φ, we may rewrite $\forall \mathbf{x}.\varphi$ to $\forall \mathbf{xy}.\varphi'$, where φ' is the result of replacing $\forall \mathbf{y}.\psi$ by ψ. If there exists a $\forall \mathbf{y}.\psi$ that has negative polarity in φ, we may rewrite $\forall \mathbf{x}.\varphi$ to $\forall \mathbf{x}.\varphi'$, where φ' is the result of replacing $\forall \mathbf{y}.\psi$ by $\psi\{\mathbf{y} \rightarrow \mathbf{f}(\mathbf{x})\}$, where $\mathbf{f}$ are fresh function symbols.

A.5 Convert to a Set of Clauses

In this section, we described how φ is converted into a set of clauses F_0, where each clause in F_0 is either ground, or an equivalence of the form $a \Leftrightarrow \forall \mathbf{x}\, \varphi$, where all other occurrences of a in F_0 have positive polarity.

Let U be a set of unprocessed formulas, initially $\{\varphi\}$, and let U' be the empty set. For each $\psi \in U$, we first apply techniques from A.1- A.4 to ψ. We then replace all occurrences of formulas $\forall \mathbf{x}\, \theta$ within ψ with fresh Boolean variables a. For each such variable a, we add $a \Leftrightarrow \forall \mathbf{x}\, \theta$ to F_0, and $(a \vee \neg\theta[\mathbf{c}/\mathbf{x}])$ to U', where $\mathbf{c}$ are fresh constants. We then convert ψ to clause normal form, obtaining the clauses F. For each clause in F containing the negation of some a where $a \Leftrightarrow \forall \mathbf{x}\, \theta \in F_0$, we replace $\neg a$ with $\neg\theta[\mathbf{d}/\mathbf{x}]$, where $\mathbf{d}$ are fresh constants, and add the resulting formula to U'. We add all other clauses in F to F_0. If U' is non-empty, we repeat the process with $U = U'$.

Example 32 *Say our input φ is $(\forall x.\exists y.P(x, y)) \Leftrightarrow Q$, where $\Leftrightarrow$ denotes iff and Q is a unary predicate. After rewriting this formula to $(\forall x.\neg\forall y.\neg P(x, y)) \Leftrightarrow Q$, we eliminate nested quantifiers to obtain $\varphi' = (\forall x.P(x, f(x)) \Leftrightarrow Q$ where f is a fresh function symbol. When processing the set $U = \{\varphi'\}$, we introduce the variable a for $\forall x.P(x, f(x))$, add $(a \Leftrightarrow \forall x.P(x, f(x)))$ to F_0, add $(a \vee \neg P(c_1, f(c_1)))$ to U' for fresh constant c_1, and convert $(a \Leftrightarrow Q)$ to clause normal form, obtaining the clauses $a \vee \neg Q$, and $\neg a \vee Q$. We replace $\neg a$ in the second clause with $\neg P(c_2, f(c_2))$, where c_2 is a fresh constant. In the end, we obtain the set $F_0 = \{a \vee \neg Q,\ \neg P(c_2, f(c_2)) \vee Q,$ $a \vee \neg P(c_1, f(c_1)),\ a \Leftrightarrow \forall x.P(x, f(x))\}$.*

APPENDIX B

EXTENSIONS TO MODEL-BASED QUANTIFIER INSTANTIATION

In this section, we mention further extensions of our model-based quantifier instantiation algorithm for computing the interpretation of terms that are not model-checkable (see Definition 11).

We can compute interpretation for inequalities over at most one integer variable. For instance, if t is model-checkable and $D_{\lambda \mathbf{x}.t}$ is definition $\mathbf{c_1} \rightarrow v_1, \ldots, \mathbf{c_n} \rightarrow v_n$, then $D_{\lambda \mathbf{x}.x_i \leq t}$ is the definition $\mathbf{c_1} \triangle_i [-\infty, v_1] \rightarrow \mathbf{true} \cdot \mathbf{c_1} \triangle_i [v_1+1, \infty] \rightarrow \mathbf{false} \cdot \ldots \cdot \mathbf{c_n} \triangle_i [-\infty, v_n] \rightarrow \mathbf{true} \cdot \mathbf{c_n} \triangle_i [v_n+1, \infty] \rightarrow \mathbf{false}$. Similarly, we can compute the interpretation for $D_{\lambda \mathbf{x}.x_i \geq t}$.

Example 33 *Say* $D_{\lambda \mathbf{x}.t}$ *is* $([-\infty, 4]) \rightarrow 2$, $([5, \infty]) \rightarrow 3$. *Then,* $D_{\lambda \mathbf{x}.x_1 \leq t}$ *is* $([-\infty, 2]) \rightarrow \mathbf{true}$, $([3, 4]) \rightarrow \mathbf{false}$, $([5, \infty]) \rightarrow \mathbf{false}$.

We can further relax our constraints on the shape of terms t for which we may produce $D_{\lambda \mathbf{x}.t}$ based on the following definition.

Definition 14 *A function* f_a *is* closed invertible *if and only if* (i) *it is closed under abstract values, and* (ii) *it has an inverse* f_a^{-1}.

For example, we can define a closed invertible function $\lambda x.(x+1)$ that is closed under abstract values: $\perp \mapsto \perp$, $v \mapsto v+1$, $[v_1, v_2] \mapsto [v_1+1, v_2+1]$, $* \mapsto *$, and an inverse can be defined: $\lambda x.(x-1)$. We now relax our set of constraints on the shape of terms we consider when applying our model-checking algorithm.

proc $\mathsf{ext_compose}(\mathbf{c} \to (t_1, \ldots, t_n), (d_1, \ldots, d_n) \to w) \equiv$
  if $n = 0$
    return $\mathbf{c} \to w$
  else if t_n is $f_a(x_i)$, and $\mathbf{c}.i$ is compatible with $f_a^{-1}(d_n)$
    return $\mathsf{ext_compose}(\mathbf{c} \triangle_i f_a^{-1}(d_n), (t_1, \ldots, t_{n-1}), (d_1, \ldots, d_{n-1}))$
  else if t_n is w, and w is compatible with d_n
    return $\mathsf{ext_compose}(\mathbf{c}, (t_1, \ldots, t_{n-1}), (d_1, \ldots, d_{n-1}))$
  else
    return $\perp \to w$
  end

Figure B.1. Extended method for computing composition of entries. Term t_i is either a value or a closed invertible function applied to a variable from $\mathbf{x} = (x_1, \ldots x_m)$ where d_i has the sort of t_i for each $i = 1, \ldots n$, and $\mathbf{c}$ is an m-tuple where $\mathbf{c}.j$ has the sort of x_j for each $j = 1, \ldots, m$.

Definition 15 *A term t is extended model-checkable if and only if (i) t is $f_a(x_i)$, where f_a is a closed invertible function, (ii) t is $f(t_1, \ldots, t_n)$, f is uninterpreted, $t_1 \ldots t_n$ are model-checkable, or (iii) t is $f(t_1, \ldots, t_n)$, and $t_1 \ldots t_n$ are model-checkable of type (ii) or (iii).*

Since the identity function is a closed invertible function, all model-checkable terms are also model-checkable terms. When computing definitions $D_{\lambda \mathbf{x}.t}$ for extended model-checkable terms t, we require a method for composing entries that accounts for when applications of closed invertible functions f_a to variables x_i occur in the range of definitions we produce. This method, $\mathsf{ext_compose}$ is given in Figure B.1. We will write $(\mathbf{c} \to \mathbf{t}) \circ (\mathbf{d} \to v)$ to refer to the entry returned by $\mathsf{ext_compose}(\mathbf{c} \to \mathbf{t}, \mathbf{d} \to v)$.

Example 34 $(*) \to (x_1 + 2) \circ ([3, 4]) \to v$ *is equal to* $([1, 2]) \to v$.

Example 35 $(*, [2, 8]) \to (x_2 - 2, x_1) \circ ([5, 9], [3, 4]) \to w$ *is equal to* $([3, 4], [7, 8]) \to w$.

Lemma 10 *If* $\mathsf{ext_compose}(\mathbf{c}, \mathbf{t}, \mathbf{d})$ *returns* $\mathbf{w}$*, then* $\gamma(\mathbf{w})[\mathbf{x}]$ *is equivalent to* $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{t}]$ *in the theory of equality.*

Proof: Using induction on n, the argument is similar to the proof of Lemma 8. The only difference is when t_n is $f_a(x_i)$, where f_a is a closed invertible function. In this case, by the induction hypothesis, the method returns a $\mathbf{w}$ such that $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c} \triangle_i f_a^{-1}(d_n))[\mathbf{x}] \wedge \gamma((d_1, \ldots, d_{n-1}))[(t_1, \ldots, t_{n-1})]$. We have that $\gamma(\mathbf{c} \triangle_i f_a^{-1}(d_n))[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(f_a^{-1}(d_n))[x_i]$, which is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(d_n)[f_a(x_i)]$. Since t_n is $f_a(x_i)$, we have that $\gamma(\mathbf{w})[\mathbf{x}]$ is equivalent to $\gamma(\mathbf{c})[\mathbf{x}] \wedge \gamma(\mathbf{d})[\mathbf{t}]$. □

We assume the composition of definitions can be computed using the extended composition of entries using the same method as the one from Section 5.4.2. We demonstrate this in the following example.

Example 36 *Say we have that* D_P *is* $([-\infty, 3]) \rightarrow \mathbf{false}$*,* $([4, 8]) \rightarrow \mathbf{true}$*,* $([9, \infty]) \rightarrow \mathbf{false}$*. Then,* $D_{\lambda \mathbf{x}.P(x+2)}$ *is* $([-\infty, 1]) \rightarrow \mathbf{false}$*,* $([2, 6]) \rightarrow \mathbf{true}$*,* $([7, \infty]) \rightarrow \mathbf{false}$*.*

The method for constructing definitions $D_{\lambda \mathbf{x}.t}$ for extended model-checkable terms t is nearly identical to the methods described in Section 5.4.2.1, with only minor modifications. First, when t is $f_a(x_i)$, we return the definition $* \rightarrow f_a(x_i)$ for $D_{\lambda \mathbf{x}.t}$. Second, we assume that composition of definitions is constructed using the extended method for composition of entries as given in Figure B.1. Due to Lemma 10, our construction of $D_{\lambda \mathbf{x}.t}$ in this extension is correct using the same argument as the one from Theorem 5.

REFERENCES

[1] W. Ackermann. *Solvable cases of the decision problem.* North–Holland, 1954.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of CAV'11*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *Proceedings of LPAR'06*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.

[5] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[6] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. $ME(LIA)$ – model evolution with linear integer arithmetic constraints. Technical Report 08-06, Department of Computer Science, University of Iowa, 2006.

[7] Nikolaj Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Proceedings of the 5th international conference on Automated Reasoning*, IJCAR'10, pages 316–330, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Børner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[9] Jasmin Christian Blanchette and Alexander Krauss. Monotonicity inference for higher-order formulas. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 91–106. Springer Berlin Heidelberg, 2010.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer Berlin Heidelberg, 2006.

[11] R. Brummayer. *Efficient SMT Solving for Bit-vectors and the Extensional Theory of Arrays*. Schriften der Johannes-Kepler-Universität Linz. Trauner, 2009.

[12] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. *AMAI*, 55(1-2):63–99, 2009.

[13] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, pages 248–254, 2012.

[14] Koen Claessen and Ann Lillieström. Automated inference of finite unsatisfiability. In RenateA. Schmidt, editor, *Automated Deduction CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 388–403. Springer Berlin Heidelberg, 2009.

[15] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. Sort it out with monotonicity: translating between many-sorted and unsorted first-order logic. In *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, pages 207–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model building. In *CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications*, pages 11–27, 2003.

[17] Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[18] Leonardo de Moura and Nikolaj Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Leonardo de Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 2010.

[21] Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, Rome, Italy, 2013*, 2013.

[22] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *Proceedings of CADE-23*, volume 6803 of *LNCS*, pages 222–236. Springer, 2011.

[23] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

[24] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[25] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[26] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[27] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE-21), Bremen, Germany*, volume 4603 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2007.

[28] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[29] P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, and M.Y. Levin. Automating software testing using program analysis. *Software, IEEE*, 25(5):30–37, Sept.-Oct.

[30] Amit Goel, Sava Krstić, and Alexander Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 12–17, New York, NY, USA, 2008. ACM.

[31] Amit Goel, Sava Krstić, Rebekah Leslie, and Mark Tuttle. SMT-based system verification with DVF. In *Proceedings of SMT'12*, 2012.

[32] Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj Bjrner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer Berlin Heidelberg, 2011.

[33] Dejan Jovanović and Clark Barrett. Sharing is caring: Combination of theories. *Frontiers of Combining Systems*, pages 195–210, 2011.

[34] Dejan Jovanović and Leonardo de Moura. Solving non-linear arithmetic. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.

[35] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[36] K. Korovin and C. Sticksel. iProver-Eq: An instantiation-based theorem prover with equality. In J. Giesl and R. Hähnle, editors, *5th International Joint Conference, IJCAR 2010*, volume 6173 of *Lecture Notes in Computer Science*, pages 196–202. Springer, 2010.

[37] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic (CSL'07)*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

[38] Konstantin Korovin. Inst-gen - a modular approach to instantiation-based automated reasoning.

[39] Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic. In *Proceedings of IJCAR'08*, volume 5195 of *LNCS*, pages 292–298. Springer, 2008.

[40] Konstantin Korovin. Non-cyclic sorts for first-order satisfiability. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pages 214–228. Springer Berlin Heidelberg, 2013.

[41] Sava Krstić and Amit Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *Proceeding of FroCoS'07*, volume 4720 of *LNCS*, pages 1–27. Springer, 2007.

[42] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *CAV*, pages 475–478, 2004.

[43] William McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, 1994.

[44] K. L. Mcmillan. Interpolation and SAT-based model checking. pages 1–13. Springer, 2003.

[45] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to richer logics. In *CAV*, pages 462–476, 2009.

[46] K.L. McMillan. Interpolants from Z3 proofs. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 19 –27, 30 2011-nov. 2 2011.

[47] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[48] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[49] Robert Nieuwenhuis and Albert Oliveras. Fast Congruence Closure and Extensions. *Inf. Comput.*, 2005(4):557–580, 2007.

[50] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[51] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving, 1999.

[52] Lawrence C Paulson and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[53] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite model finding in SMT. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer Berlin Heidelberg, 2013.

[54] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In Maria Paola Bonacina, editor, *Automated Deduction CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin Heidelberg, 2013.

[55] Andrew Reynolds, Cesare Tinelli, and Liana Hadarean. Certified interpolant generation for EUF. In S. Lahiri and S. Seshia, editors, *Proceedings of the 9th International Workshop on Satisfiability Modulo Theories*, 2011.

[56] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[57] Geoff 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.

[58] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Proceeding of FroCoS'96*, Applied Logic, pages 103–120. Kluwer Academic Publishers, 1996.

[59] Mark R. Tuttle and Amit Goel. Protocol proof checking simplified with SMT. In *Proceedings of NCA'12*, pages 195–202. IEEE Computer Society, 2012.

[60] Hantao Zhang and Mark E. Stickel. An efficient algorithm for unit propagation. In *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH96), Fort Lauderdale (Florida USA*, pages 166–169, 1996.

[61] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI95)*, 1995.

[62] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 17–36, London, UK, UK, 2002. Springer-Verlag.