Disproving in First-Order Logic with Definitions, Arithmetic and Finite Domains

Joshua Bax

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

November 2017

Except where otherwise indicated, this thesis consists of my original work.

Joshua Bax
23 November 2017

For my parents.

Acknowledgments

First of all, thanks to my supervisor Peter Baumgartner. You were always around to answer my questions and this thesis is in the end, the result of our many discussions. Also I'd like to thank the other members of my panel, Professor John Slaney and Professor Phil Kilby. Thanks also to Uwe Waldmann, for allowing me to share in your research.

Thanks to my office mates and fellow PhD students Jan Kuester and Mohammad Abdulaziz for helping me to understand this business, and for shooting down many stupid ideas late on Friday afternoons. Bruno Wolzenlogel-Paleo was also included in this triage of ideas.

Thanks to Geoff Suttcliffe and Christian Suttner for their efforts on the TPTP library, as well the CASC competition. Without any of those this would have been a brief thesis. Thanks to the CADE conference organizers for monetary support also.

Thanks to Data61 née NICTA, for generous funding, and thanks to the many people there who provided both inspiration and support. I'm proud to have you all as colleagues.

Lastly, thanks to my parents and An Ran for their unwavering support throughout. I really couldn't have done it without you. Those whom I may have missed, rest assured you have my gratitdue.

Abstract

This thesis explores several methods which enable a first-order reasoner to conclude satisfiability of a formula modulo an arithmetic theory. The most general method requires restricting certain quantifiers to range over finite sets; such assumptions are common in the software verification setting. In addition, the use of first-order reasoning allows for an implicit representation of those finite sets, which can avoid scalability problems that affect other quantified reasoning methods. These new techniques form a useful complement to existing methods that are primarily aimed at proving validity.

The Superposition calculus for hierarchic theory combinations provides a basis for reasoning modulo theories in a first-order setting. The recent account of 'weak abstraction' and related improvements make an implementation of the calculus practical. Also, for several logical theories of interest Superposition is an effective decision procedure for the quantifier free fragment.

The first contribution is an implementation of that calculus (Beagle), including an optimized implementation of Cooper's algorithm for quantifier elimination in the theory of linear integer arithmetic. This includes a novel means of extracting values for quantified variables in satisfiable integer problems. Beagle won an efficiency award at CADE Automated theorem prover System Competition (CASC)-J7, and won the arithmetic non-theorem category at CASC-25. This implementation is the start point for solving the 'disproving with theories' problem.

Some hypotheses can be disproved by showing that, together with axioms the hypothesis is unsatisfiable. Often this is relative to other axioms that enrich a base theory by defining new functions. In that case, the disproof is contingent on the satisfiability of the enrichment.

Satisfiability in this context is undecidable. Instead, general characterizations of definition formulas, which do not alter the satisfiability status of the main axioms, are given. These general criteria apply to recursive definitions, definitions over lists, and to arrays. This allows proving some non-theorems which are otherwise intractable, and justifies similar disproofs of non-linear arithmetic formulas.

When the hypothesis is contingently true, disproof requires proving existence of a model. If the Superposition calculus saturates a clause set, then a model exists, but only when the clause set satisfies a completeness criterion. This requires each instance of an uninterpreted, theory-sorted term to have a definition in terms of theory symbols.

The second contribution is a procedure that creates such definitions, given that a subset of quantifiers range over finite sets. Definitions are produced in a counter-example driven way via a sequence of over and under approximations to the clause set. Two descriptions of the method are given: the first uses the component solver

modularly, but has an inefficient counter-example heuristic. The second is more general, correcting many of the inefficiencies of the first, yet it requires tracking clauses through a proof. This latter method is shown to apply also to lists and to problems with unbounded quantifiers.

Together, these tools give new ways for applying successful first-order reasoning methods to problems involving interpreted theories.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

Formalizations of problems in software verification typically involve quantification, equality, and arithmetic. The Satisfiability Modulo Theories (SMT) field has made significant progress in developing efficient solvers for such problems, but solvers for first-order logic have yet to catch up, despite a strong base of equational and quantifier reasoning capability. The reason for this is the high theoretical complexity of reasoning with interpreted theories, specifically arithmetic. SMT-solvers also have the valuable capability of providing counter-example models, while first-order solvers are better able to produce proofs of valid theorems.

The Superposition calculus for hierarchic theory combinations provides a sound basis for reasoning modulo theories in a first-order setting. The recent account of 'weak abstraction' and related improvements make an implementation of the calculus practical. Also, for several logical theories of interest, Superposition is an effective decision procedure for the quantifier free fragment.

This thesis explores several methods which enable a first-order reasoner to conclude satisfiability of a formula, modulo an arithmetic theory. The most general method requires that certain quantifiers are restricted to range over finite sets, however, such assumptions are common in the software verification setting. Moreover, the use of first-order reasoning allows for an implicit representation of those finite sets, possibly avoiding scalability problems that affect other quantifier reasoning methods. These new techniques will form a useful complement to existing methods usually aimed at proving validity.

1.2 Introduction

The most successful verification technologies today, measured in terms of their use in practical applications to industrial problems, are those of Constraint Programming (CP) and of SMT. These are routinely used for solving difficult optimization and software verification problems.

Using this metric, it appears that the state-of-the-art in first-order theorem proving lags behind. The main technical reason for that is the inherent difficulty in

combining reasoning for *quantified* first-order formulas with reasoning for specialized background theories (theorem proving in this general setting is not even semi-decidable). The CP and SMT approaches avoid this issue by dealing with quantifier-free (ground) formulas only, but doing so in a very efficient way.

Not being able to deal with quantified formulas is a serious practical limitation; it limits the range of potential applications, if not the scale. This has been recognized for software verification applications, but currently the limitation is addressed in an ad hoc way: all SMT approaches today rely on heuristic instantiation of quantifiers to deal with quantified formulas. The drawback to heuristic instantiation is that completeness can be guaranteed only in very limited cases. Consequently, such methods will often not find proofs in expected cases, and are not well suited for disproving invalid conjectures stemming from buggy programs. On the other hand, first-order theorem proving approaches inherently support reasoning with quantified formulas, but lag behind in reasoning with background theories for the reasons mentioned above.

The main hypothesis of this research is that by combining and advancing recent developments in first-order theorem proving, as well as ideas from the CP and SMT fields, it will be possible to design theorem provers that better support reasoning with quantified formulas *and* background theories together.

1.3 Thesis Outline

Chapter 2 introduces the conventions used in the thesis as well as the Hierarchic Superposition calculus, which is the main tool for first-order reasoning used later. Several first-order theories of interest for software verification are identified. Chapter 3 describes *Beagle* , a test-bed implementation of Hierarchic Superposition with Weak Abstraction. This includes an optimized implementation of Cooper's algorithm capable of returning solutions to quantifier free problems in the form of bindings to the free variables. Some useful parametric test problems for integer arithmetic are defined and experiments with the customized arithmetic solver are reported on. Chapter 4 describes a method for classifying problems in which theories (in particular integer and other infinite theories) are extended with new definitions in such a way that satisfiability is not compromised. This allows using a refutation-based solver (i. e. , one only capable of showing unsatisfiability of formula sets) to show satisfiability of a given conjecture by showing that its negation is contradictory. Chapter 5 follows the theme of disproving and considers the case where a given hypothesis and its negative are satisfiable. Being theoretically more difficult, this task requires stronger assumptions on the input clause set. The method presented in the chapter assumes that a subset of quantifiers in the input are restricted to range over finite integer sets in such a way that the number of ground instances of free theory sorted terms is finite. The method proceeds by sequential under and over approximation in order to limit generation of new clauses instances. This allows concluding T-satisfiability of certain clause sets; otherwise impossible due to the fact that the semantics of Hierar-

chic Superposition allows degenerate models. An advantage of this first method is that it can be implemented with off-the-shelf solvers, at the cost of some inefficiency in computing successive over-approximations. Chapter 6 expands on the method in Chapter 5 by considering the equations that define the over-approximation separately from the clause set. This eliminates some built-in inefficiencies of the prior method and enables an analysis of just which changes are required to advance to the next over-approximation set. The analysis is automatically done with a small prover modification and new heuristics to minimize the resulting change set are given. The abstract description of the method in Chapter 5 can be carried over to create an analogous method that works for recursive data structures and a method for unbounded domains is sketched. The class of basic definitions is defined, which can be excluded from the approximation process yielding further reductions in instantiation. Experiments show that heuristics enabled by the more general description do lead to an increase in performance of the definition search.

1.3.1 Joint Contributions

Each chapter except this one and the last are based on papers authored with other people. This section outlines the extent of the reuse of the paper content in each chapter, as well as the new contributions.

Chapter 4 is based on Baumgartner and Bax [BB13]. That chapter provides more general theorems (new) relative to the original results and describes more applications.

Chapter 3 is based on the system description in Baumgartner et al. [BBW15]. Much of the theoretical work in developing *Beagle* is due to Baumgartner and Waldmann, and work on the implementation of *Beagle* is joint work with Baumgartner. The section on solution finding and on LIA examples are original.

Chapter 5 is based on Baumgartner et al. [BBW14]. The original idea underpinning this version of the checkSAT algorithm is due to Baumgartner. I contributed an implementation as well as experiments to that paper. The presentation of the algorithm in the chapter is new, as are most of the proofs. As said above, Chapter 6 consists of original work.

1.4 An Overview of Automated Reasoning

1.4.1 Constraint Solving and SAT

A constraint satisfaction problem is given as a set of variables each with an associated domain of solution and a set of relations (the constraints) over the variables or subsets thereof. An assignment of values to variables solves the constraint problem when each set of variables satisfies its constraint. This framework can be used to encode many problems, such as scheduling/routing problems, planning, and various other optimization problems. It generalizes Linear Programming problems as it allows for

arbitrary relations over arbitrary sets of objects. A concrete definition is given by Dechter [Dec03], as well as an overview of current algorithms and search strategies.

Constraint solving formalizes many common aspects of reasoning problems and this abstract framework allows the development of generic approaches which can be applied to a diverse range of problems. At its core, constraint solving focuses on search strategy and inference. The basic search strategy is *backtracking* and most other methods are essentially refinements of this, for example, lookahead and look-back fall into this category. Inference in constraint solving tries to reduce the search space based on the structure of the problem. The key inference process in constraint satisfaction is called *constraint propagation*. In this process the constraints are inspected and the domains of variables are restricted in order to enforce varying degrees of *consistency*; arc-consistency is the weakest– requiring only that an admissible value for a variable is also admissible in every constraint over that variable. Path-consistency is stronger, it requires that any assignment to a pair of variables can be extended to a full solution.

For certain combinations of constraints a higher level of consistency is desired: a common example is when there are many mutual disequations, for example, when assigning drivers to buses, no driver simultaneously drives two buses. Enforcing low-level consistency on the variables participating in these constraints often does not produce any useful inferences, and enforcing high-level consistency on the problem is costly. Instead, a middle ground is struck and consistency is enforced only for the disequation constraints. The regularity of these constraints yields an efficient algorithm for consistency based on matchings on bipartite graphs. Constraint solvers provide a modelling abstraction called an *allDifferent* constraint that allows replacing the individual disequations with a single equivalent constraint. The new constraint can be solved using a specialized algorithm. In general many such constructions have been given for problems such as solving weighted sums or bin packing problems, these are known as *global constraints*.

Finite constraint satisfaction problems have been shown to fall in the fragment of *effectively propositional* (EPR) formulas of first-order logic [Mac92]. Formulas in the EPR fragment are equivalent to (possibly very large) finite sets of ground formulas– hence they can be solved by a SAT/SMT solver. Conversely, SAT is itself a specific instance of a constraint satisfaction problem. The fact that the two approaches can be translated between in no way implies such a translation would be efficient, and there are other respective benefits to the two approaches besides. A good comparison of SAT with CP can be found in Bordeaux et al. [BHZ05].

SAT also focuses on both search and inference, although the language and data structures it uses are much more restricted than that found in CP problems. In CP, problems are modelled directly, often natively in the programming language of the library– most problems treated by SAT solvers are translated via a specialized tool into propositional formulas. Furthermore, SAT is a push-button/black box technology focused on verification only, while CP is mostly open and programmable and can do optimization as well as verification.

Many works have suggested the application of CP techniques to theorem provers.

Sometimes the use is explicit: as a theory solver for SMT [Nie10], otherwise the use is implicit, specifically using the underlying algorithms to improve various reasoning tasks [BS09, Mac92]. This was a motivation for the Finite Model finding technique, which is described below.

1.4.2 Superposition and First-Order theorem proving

First-order logic theorem proving aims to develop push-button verification technology for recognizing first-order logic theorems. First-order logic is expressive enough to encode almost all modern mathematics and, as would be expected, is undecidable. However, classical results also show that the set of valid formulas of first-order logic is in fact enumerable– this follows from the existence of effective calculi whose rules produce valid formulas from a set of axioms. From Herbrand's theorem and the compactness theorem it follows that any given unsatisfiable formula can be demonstrated as such by finding a finite set of *instances* of the formula which do not have a model. It is this result that underlies automated theorem proving and defines its limitations: if the given formula is not valid the solver may never be able to prove it so. So first-order theorem provers are at best semi-decision procedures for first-order logic.

First-order proof calculi designed specifically for automation had their origin with Robinson's Resolution method in the early 60's [Rob65b], followed by the introduction of the Paramodulation calculus [RW69], which introduced a dedicated inference rule to deal with identity. Later refinements used orderings to restrict the proof search; methods extending the Knuth-Bendix completion method to first-order logic are known as Superposition calculi [BG98, NR01] and are generally considered to be state-of-the-art when theorem proving over equational theories.

The resolution calculus comprises the following basic rules:

$$\text{Resolution}\ \frac{L \vee C \qquad \neg M \vee D}{(C \vee D)\sigma}$$

$$\text{Factoring}\ \frac{L \vee M \vee D}{D\sigma}$$

in both cases σ is a most general unifier of L and M.

These rules are applied to an input formula in conjunctive normal form (it is well-known that all first-order formulas have an equivalent CNF formula, and that such normal forms can be effectively found). Key properties that this calculus enjoys are *soundness*: all conclusions of inference rules are logical consequences of their premisses; and *refutational completeness*: if the input formula is a valid theorem then the proof procedure will eventually confirm this. The calculus works in a refutational setting, meaning that when given a theorem consisting of hypotheses and conjecture to confirm the calculus demonstrates this by proving the unsatisfiability of the *negated* conjecture given the hypotheses.

Though meeting with some early success, it was soon recognised that resolution had severe shortcomings when using theories involving equality. In particular, resolution on the transitivity axiom and reflexivity axioms produce infinitely many new clauses. The Paramodulation calculus was developed to deal more effectively with equality. It adds a new inference rule to the resolution calculus, which encodes the familiar 'replace like by like' rule for equality:

$$\text{Paramodulation}\ \frac{C \vee s \approx t \qquad D[u]}{(D[s] \vee C)\sigma}$$

where σ is the mgu of t and u

However, the Paramodulation calculus fell short of its goal– it was found that the paramodulation rule still produced too many irrelevant clauses to be useful.

An approach that helped to push forward development of first-order theorem proving is Knuth-Bendix completion [KB83]. Essentially, it is a method for transforming a set of equations into a new, ordered set of equations which, when applied non-deterministically, constitute a decision procedure for the word problem of the original algebra.

This method was generalised from single ground equations to full first-order clause logic, and the orderings used in the Knuth-Bendix method were found to confer a strong enough restriction on the productivity of the paramodulation rule to make it practically useful. This development is reviewed in Bachmair and Ganzinger [BG98]. The combination of the paramodulation rule along with the ordering strategy of Knuth-Bendix completion is known as Superposition.

$$\text{Positive-Superposition}\ \frac{C \vee s \approx t \qquad D \vee u[s'] \approx v}{(C \vee D \vee u[t] \approx v)\sigma}$$

where $u\sigma \not\succeq v\sigma$ and $s\sigma \not\succeq t\sigma$, $(s \approx t)\sigma$; $(u \approx v)\sigma$ are maximal in their respective clauses,$(s \approx t)\sigma \not\succeq (u \approx v)\sigma$ and s' is not a variable.

This is an example of the superposition rule, there is a corresponding rule for superposition on negative equations, as well as rules for resolution with the reflexivity axiom, and for factoring (as in the resolution calculus). The ordering $\succeq$ is extended from terms, and must be invariant under substitution and under contexts (i. e. , ,$s \succeq t$ implies $u[s] \succeq u[t]$ for any u). This is exactly the ordering used in Knuth-Bendix completion to ensure that the generated rewrite system is convergent. Several orderings satisfy these criteria, and each exhibits different performance characteristics. Most modern first-order theorem provers are based on this method, such as E [Sch04], SPASS [WSH$^+$07], Vampire [RV01], and Waldmeister [BH96].

1.4.3 First-order theorem proving with Theories

It was recognised some time ago, that the efficiency of reasoning could be improved by incorporating knowledge about existing theories into the reasoning process. Initially first-order theorem proving aimed to address the question of satisfiability of the input formulas in full generality, though it is hardly useful when one is extending a particular theory with a fixed interpretation– say that of lists or of linear arithmetic. This approach would add theory axioms to the input and apply the proof procedure to the extended set of formulas. The problem is then, that it allows non-standard interpretations of the theory in question, and it is impossible when the theory is not finitely axiomatizable.

Stickel [Sti85] describes *Theory Resolution*, which generalizes the resolution calculus by allowing a resolution inference on clause literals which are possibly not syntactic complements but are complements modulo the theory in question. For example, $1 < x$ and $x - 1 < 0$ do not unify in the standard sense but together are unsatisfiable in the theory of arithmetic. It is also shown that this is a generalization of the Paramodulation calculus (theory resolution with the theory of uninterpreted functions and equality). However, to admit this generalization one needs the ability to compute all potential *theory unifiers* within the given theory– of which there may be infinitely many. For example, unifying $p(x + y)$ and $\neg p(11)$ yields $[x \mapsto 1, y \mapsto 10], [x \mapsto 2, y \mapsto 9] \ldots$.

Bürckert [Bür94] gives a thorough treatment of this problem for the resolution calculus. Theory literals are removed from clauses by a process of abstraction and added to a constraint subclause. Effectively, each clause C is translated to a logically equivalent formula $D \rightarrow E$, where D is a conjunction of theory literals only, and E is a disjunction of strictly non-theory literals. Resolution is performed on the non-theory part of clauses, and constraints are accumulated until a (not-necessarily unique) constrained empty clause is derived. The constraint (a conjunction of non-ground theory literals) is checked by an appropriate theory solver, and if the constraint is satisfiable, the satisfying model is removed from consideration. Once all possible models have been eliminated in this way, the proof search terminates. The advantage of this approach is that all theory literals are excluded from the proof search, drastically reducing the search space. A shortcoming is that too much is assumed of the theory solver, in particular, when combining the theory of equality of uninterpreted functions with other theories (the theory solver should be able to deal with any user defined functions which range into the other theories). However, this is typically not the case, as the theory of the solver is usually fixed.

A more workable approach is that of Bachmair et al. [BGW94], who use the framework of Hierarchical Specifications to allow a Superposition calculus to conservatively extend fixed theories. This is the subject of Chapter 2.

Some inroads have been made to addressing the problem of incorporating specialised theory reasoning for first-order solvers, but the problem is far from solved.

1.4.4 Satisfiability Modulo Theories

SMT-solvers combine SAT solving with dedicated theory decision procedures. An introduction to SMT solvers can be found in Barrett et al. [BSST09], while a good summary of the main decision procedures used in SMT solving is in Bradley and Manna [BM07].

The decoupling of solvers is advantageous, as it allows both the SAT solver and the theory solver to be pluggable. So a single SMT implementation can be updated to use the latest SAT technology, as well as support multiple theories within a common reasoning framework. Theory decision procedures must decide satisfiability of conjunctions of ground literals within the language of the theory. Classic decision procedures include those for theories of Equality, Linear Integer Arithmetic, Arrays, Lists and other inductive data types, and fixed-width bitvectors.

Commonly, verification tasks involve several of the above theories in combination. The Nelson-Oppen procedure [NO79] is used to allow decision procedures for separate theories to cooperate, together deciding satisfiability for conjunctions of ground literals in the combined theory. This procedure requires that the signatures of the background theories are disjoint, and may not be possible where one of the theories is finite.

The restriction to ground formulas imposed by decision procedures for the individual theories gives good performance guarantees, but means that quantified formulas must be reduced to ground formulas by other means. In general, formulas $\forall x.\, F[x]$ are successively ground instantiated with theory terms until an unsatisfiable set of instances of F is found. Various methods [GBT07, GdM09, DNS03] have been proposed that identify instances to use for instantiation, or quantified fragments of theory languages which admit instantiation to a set of equi-satisfiable ground instances. SMT solvers are normally used as part of larger verification environments such as Isabelle/HOL or ACL2, or as part of specification languages which manage the translation of verification conditions into complete fragments. Examples of such specification languages are Boogie [BCD+05] and Why3 [FP13], which support full-blown implementations of verification languages (e. g. , Dafny [Lei10] and Frama-C [KKP+15], respectively), and interface with SMT solvers to discharge their verification conditions.

Some work has been done towards using constraint solvers (particularly specialised propagators), as theory solvers with this method as well as applying more general ideas from constraint solving to SMT and SAT. For example, Nieuwenhuis [Nie10] suggests this, as well as using more general CP heuristics in the SAT part of the SMT procedure. Conversely, it has been noted that CS problems are instances of SMT problems, and so SMT techniques can be applied in that direction too [NOT06, BPSV12]. First-order theorem provers have also been adapted to work as theory solvers for SMT, in particular, superposition based methods are investigated by Armando et al. [ABRS09] as theory solvers for many theories, with criteria for combinations of theories also described.

More detail on SMT solving is given in Chapter 2.

1.5 Summary

Theorem proving over bounded domains with arithmetic is a difficult problem class with applications in bounded model checking, formal mathematics, and other verification tasks. It can be translated from first-order logic to a constraint satisfaction or SAT problem (for which efficient tools exist), however, a direct translation from one to the other is often inefficient. SMT solvers may also encounter problems when instantiating quantifiers in the original problem. A different approach is to take a first-order solver based on the calculus given in Baumgartner and Waldmann [BW13b] and modify it to produce a decision procedure for the restricted case considered. While simple cases have been treated (or fall under existing generalizations which have strong assumptions), an important question still remains: how can *new* functions be added to the theory without losing decidability (for finite domains), or completeness in the general case? This is illustrated in the following simple example.

Example 1.5.1 (Loss of Completeness for Introduced Functions)**.** Let f_4 be a new operator symbol that maps integers to integers. It will represent a permutation on $\{1,2,3,4\} \subset \mathbb{Z}$. The assumption that this is a subset of integers will entail that the elements are distinct. The following formulas assert that f_4 is a permutation on the appropriate set:

$$\forall x : \mathbb{Z}, y : \mathbb{Z}.\ ((1 \leq x \wedge x \leq 4 \wedge 1 \leq y \wedge y \leq 4) \Rightarrow (f_4(x) \approx f_4(y) \Rightarrow x \approx y))$$
$$\forall x : \mathbb{Z}.\ (1 \leq x \wedge x \leq 4) \Rightarrow \exists y : \mathbb{Z}.\ (1 \leq y \wedge y \leq 4) \wedge (f_4(y) \approx x)$$

The goal will be to show that if 1 and 2 are members of 2-cycles and $f_4(3) = 3$ then $f_4(4) = 4$, i. e. , f_4 is $(1,2)$ in cycle notation. Formally, the solver must show that the formula

$$(f_4(f_4(1)) \approx 1 \wedge f_4(f_4(2)) \approx 2 \wedge f_4(3) \approx 3) \wedge f_4(4) \not\approx 4$$

is inconsistent relative to the above formulas and the integer theory. An implementation of the Superposition calculus for Hierarchic combinations of theories would fail to derive a contradiction in this case. This is because the term $f_4(4)$ is not identified with any integer ($f_4(3)$ and $f_4(f_4(1))$ are, however). Thus, the surjectivity axiom does not enforce $f_4(4) \in \{1,2,3,4\}$, and no contradiction follows.

Chapter 2

Background and Related Work

2.1 Motivation

This chapter covers the definitions and lemmas required in each chapter of the thesis. Section 2.2 gives an account of first-order logic syntax and semantics, specifically monomorphic (sorted) equational logic. Section 2.3 describes some logical theories. The additive theory of integers is used as an interpreted theory in most examples while the data structure theories are used as a source of problems and to provide context for applications. Section 2.4 describes some common notions from saturation based proof calculi, in particular definitions of terms, term algebras and substitutions are used throughout. Section 2.5 describes a specific calculus for reasoning in hierarchic combinations of theories. This calculus will form the basis for the implementation in Chapter 3 and will be the reasoning component used in other chapters Section 2.6 describes other reasoners that use either integers or incorporate background reasoning in some way.

2.2 Syntax and Semantics

Throughout this thesis, the following standard account of first-order logic with equality will be used.

The logic is *many-sorted*: each term is assigned a sort. The type system employed is monomorphic, all of the sorts are constant symbols with no internal structure. Concretely, a many-sorted logic restricts the set of values that can be assigned to variables and restricts both terms in an equation to have the same sort. Sorts are assumed to be non-empty in every interpretation and distinct sorts are disjoint.

A *signature* Σ is a tuple (Ξ, Ω) consisting of a finite set of *sort symbols* $\Xi = \{S_1, \ldots, S_n\}$ and set of operator symbols Ω with associated *arities* over the sorts in Ξ, written $f : S_1 \times \ldots \times S_n \to S$ for example.

All signatures are a assumed to have at least the Boolean sort *Bool*, as well as the constant symbol *true*. Predicate applications, e. g. , $p(x)$, are modelled by the atomic equation $p(x) \approx \mathit{true}$ (where $\approx$ is the logical symbol for equality) and negated predicate atoms $\neg p(x)$ by $p(x) \not\approx \mathit{true}$. These are usually abbreviated to the non-equational form in the text. Only predicates and *true* have the sort *Bool*, in particular

there are no *Bool*-sorted variables.

A signature Σ is a *sub-signature* of another signature Σ', written $\Sigma \subset \Sigma'$, if all sorts and function symbols of Σ are included in Σ', with the arities of the function symbols unchanged. Then Σ' is referred to as an *extension* of Σ. In most cases, the extension signature adds function symbols only, e. g. , in Skolemization, and when the new function symbol needs to be identified the extension signature will be written $\Sigma \cup \{f\}$, for example, abbreviating $(\Xi_\Sigma, \Omega_\Sigma \cup \{f\})$.

Given a signature Σ, and a countable infinite set of variable symbols $\mathcal{X}$, such that for each $S \in \Xi$, $\mathcal{X}$ contains infinitely many variables of that sort (except *Bool* of course), then the set of Σ-terms $T(\Sigma, \mathcal{X})$ is defined inductively as:

1. Any $x \in \mathcal{X}$ or 0-ary *constant* symbol $c \in \Sigma$
2. $f(t_1, \ldots, t_n)$ for all $f : S_1 \times \ldots \times S_n \to S \in \Omega$ and all Σ-terms $t_1, \ldots, t_n$ having sorts $S_1, \ldots, S_n$ respectively.

Terms in $T(\Sigma, \emptyset)$ are called *ground terms*.

Arbitrary variables appearing in the text will be written using x, y, z; constants written a, b, c, d; and applications of function symbols f, g, h to terms: $f(s,t), g(x)$, for example. Terms in general are represented with letters $l, \ldots, t$. Boolean terms are called atomic formulas, or *atoms* for short.

Logical symbols include the usual Boolean connectives $\wedge, \vee, \neg, \Rightarrow$; quantifiers $\forall, \exists$; and equality, denoted by $\approx$. Equality ($\approx$) is not included in the signature as it is a logical symbol. As such, it is always interpreted as an equivalence relation, so the equality axioms (reflexivity, transitivity, symmetry and functional congruence) are superfluous. The symbol $=$ denotes identity of mathematical objects in meta-logical statements.

That a term t has sort S, is indicated by $t : S$ in variable lists of quantifiers or in running text. To indicate the sort of a subterm in a formula or of both terms in an equation, a subscript is used, e. g. , in $a \approx_S b$ both a and b have sort S.

Since sorts are assigned disjoint sets, terms in an equation must be of the same sort; it is assumed that all well-formed formulas satisfy this requirement (well-sortedness). The *language* of Σ is the set of all well-formed formulas made from Σ-terms.

An *interpretation* $\mathcal{I}$ of a signature Σ consists of

- a *domain* $\mathcal{D}_\mathcal{I} = \{\xi_1, \ldots, \xi_n\}$ that interprets the sorts $\Xi = \{S_1, \ldots, S_n\}$ of Σ, and
- an *assignment* which maps from function (and constant) symbols $f : S_1 \times \ldots \times S_n \to S_k$ of Ω to n-ary functions $f^\mathcal{I} : \xi_1 \times \ldots \times \xi_n \to \xi_k$.

It is required that the sorts are *inhabited*, i. e. , no ξ_i is empty and that they are pairwise disjoint.

An interpretation defines a unique map from terms in $T(\Sigma, \emptyset)$ to $\mathcal{D}_\mathcal{I}$, the image of a Σ-term under this map is $\mathcal{I}(t)$ for a Σ-interpretation $\mathcal{I}$. An interpretation $\mathcal{I}$ *satisfies* an equation $s \approx t$ iff $\mathcal{I}(s) = \mathcal{I}(t)$. Satisfaction of Boolean combinations of

ground equations is defined according to the usual truth tables. A *valuation* ν for interpretation $\mathcal{I}$ is a map from $\mathcal{X}$ to $\mathcal{D}_\mathcal{I}$. Valuations lift homomorphically to terms, atoms, and formulas; simply replacing variables consistently in their contexts. Then, an existential formula $\exists x_1, \ldots, x_k.\ F$ is satisfied if there is an interpretation $\mathcal{I}$ and valuation ν over $\mathcal{I}$ such that $\mathcal{I}$ satisfies $\nu(F)$. It is assumed that universals abbreviate negated existentials. (Note that Section 2.4 gives a slightly different semantics for clauses).

Given Σ' and sub-signature Σ, the Σ-*reduct* of a Σ'-interpretation is the unique Σ-interpretation obtained by restricting the domain to just the sorts of Σ. Since the function arities of Σ do not include sorts from Σ', the interpretation of these symbols does not change in the Σ-reduct.

A *logical theory* is simply a set of interpretations of the same signature, closed under isomorphism. The theory axiomatized by a set of Σ-formulas (the axioms) is the maximal set of Σ-interpretations that satisfy those axioms, again, closed under isomorphism. Then, given a theory T with signature Σ, a Σ-formula is *T-satisfiable* if some $\mathcal{I} \in T$ satisfies it, and *T-valid* if all $\mathcal{I} \in T$ satisfy it.

The entailment symbol '$\models$' is used in several ways in this thesis. Assume ϕ is a Σ-formula or clause, T is a theory with signature Σ, $\mathcal{I}$ is a Σ-interpretation, and $\mathcal{N}$ is a set of Σ-formulas (or clauses), then $\models$ can be used

- As shorthand for 'satisfies'. If $\mathcal{I}$ satisfies ϕ, then $\mathcal{I} \models \phi$.
- To indicate logical entailment between formulas. $\mathcal{N} \models \phi$ iff every interpretation that satisfies all formulas of $\mathcal{N}$ also satisfies ϕ.
- To indicate entailment by a theory. $T \models \phi$ iff every $\mathcal{I} \in T$ satisfies ϕ.
- To indicate logical entailment *relative* to a theory. $\mathcal{N} \models_T \phi$ iff every $\mathcal{I} \in T$ that satisfies $\mathcal{N}$ also satisfies ϕ.

The statements $T \models \bot$, and $T \models \square$ (empty clause) are shorthand for T being unsatisfiable.

The *quantifier-free fragment* of a language is the subset of the language built without quantifiers, where unbound variables are treated as if they were existentially quantified. The quantifier-free conjunctive fragment is a sub-fragment of the above which only contains conjunctions of possibly negated atoms. Some authors do not make this distinction, since any decision procedure for satisfiability of formulas in the quantifier-free conjunctive fragment can be made into a decision procedure for satisfiability in the quantifier-free fragment by transforming a given formula to disjunctive normal form and testing each disjunct in turn.

2.3 First-Order Theories for Computation

This section is based on the presentation of the theories in Bradley and Manna [BM07], however, the original axiomatization of arrays dates back to McCarthy [McC62]. The

decidability results of Nelson and Oppen [NO80] have influenced the choice of axiomatizations of recursive data structures.

2.3.1 Linear Integer Arithmetic

Presburger Arithmetic is the language of arithmetic over the natural numbers $\mathbb{N}$ without multiplication. Though lacking in expressivity, Presburger formulas arise frequently in software verification: simple while-loops can be modelled [BMS06], and both integer valued linear programming problems and constraint satisfaction problems, can be expressed with Presburger Arithmetic formulas. Furthermore, the theory allows *quantifier elimination*: each quantified formula is equivalent to a quantifier-free (ground) formula, and so the first-order theory is decidable.

The decidability of Presburger Arithmetic was shown by M. Presburger [PJ91], and the quantifier elimination procedure used today was given by Cooper [Coo72]. The latter procedure works in a language extended with multiplication and division by constant coefficients, whose theory is equivalent to Presburger Arithmetic.

The signature of Presburger Arithmetic is $\Sigma_P = \{0, s^1, +^2\}$, where 0 is a constant, s^1 is the 1-ary successor function and $+^2$ is addition, written infix. The only sort apart from *Bool* is $S_{\mathbb{N}}$, the sort of natural numbers. The axioms for Presburger arithmetic are:

$$
\begin{array}{lll}
(1) & s(x) \not\approx 0 & (2) \quad s(x) \approx s(y) \Rightarrow x \approx y \\
(3) & x + 0 \approx x & (4) \quad x + s(y) \approx s(x + y) \\
(5) & \multicolumn{2}{l}{(\phi[0] \wedge \forall n.\, (\phi[n] \Rightarrow \phi[n+1])) \Rightarrow \forall x.\, \phi[x],} \\
 & \multicolumn{2}{l}{\text{where } \phi \text{ is any } \Sigma_P\text{-formula with one free variable.}}
\end{array}
$$

The language of Presburger Arithmetic is too cumbersome for most applications in software verification. More common is *Linear Integer Arithmetic* (LIA) which has signature $\Sigma_{\mathbb{Z}} = \{\ldots, -2, -1, 0, 1, 2, \ldots, -^1, +^2, <^2\}$; the sort of integers $S_{\mathbb{Z}}$ is the only sort. The axioms of LIA are those of a linearly ordered Abelian group where $\{0, -, +, <\}$ have their expected roles. Its canonical model is $\mathbb{Z}$ with the natural addition function and order relation. The theory of LIA is equivalent to Presburger arithmetic as $\Sigma_{\mathbb{Z}}$-formulas can be directly translated to Σ_P-formulas [BM07].

In order to better support the combination with other theories, it is useful to extend $\Sigma_{\mathbb{Z}}$ with a countable infinite set Π of fresh constant symbols, called *parameters*. The theory of LIA with parameters consists of interpretations in which the operators in $\Sigma_{\mathbb{Z}}$ have their canonical interpretation and parameters in Π are always interpreted as members of $\mathbb{Z}$. A formula with parameters is equivalent to a $\Sigma_{\mathbb{Z}}$-formula where the parameters are replaced with existentially quantified variables. However, LIA with parameters is *non-compact*: consider the infinite set of formulas $\{0 < \alpha, 1 < \alpha, \ldots\}$ where α is a parameter. Every finite subset is satisfiable, but there is clearly no interpretation satisfying the entire set in the theory of LIA with parameters, as α must be interpreted as an integer. This fact will become important in Section 4.

Cooper's Algorithm: Cooper's algorithm [Coo72] for quantifier elimination in LIA (and therefore, for deciding $T_{\mathbb{Z}}$-validity of $\Sigma_{\mathbb{Z}}$-formulas) is well known. Although formulas with arbitrary quantifier structure can be checked, the complexity of the procedure is very high: Oppen [Opp78] gives an upper bound time-complexity of $2^{2^{cn}}$ for formulas of length n and some positive constant c, while Fischer and Rabin [FR74] show that for most lengths n there is some formula which will take at least $2^{2^{dn}}$ steps to check validity, for some constant d. Despite this, Cooper's algorithm has the advantage of being well understood, e. g. , optimizations are already described in Reddy and Loveland [RL78], and implementations including various optimizations are described elsewhere [Har09, PH15, BM07]. Moreover, it is advantageous to have a single algorithm that can discharge proof goals of varying complexity. Consider the use of Cooper's algorithm in the Isabelle/HOL proof environment[1]: the well-known proofs of correctness allow for a verified implementation and the algorithm's generality allows it to be used as a component solver in the proof assistant.

The relationship between complexity and quantifier structure for Peano Arithmetic formulas has also been investigated. Reddy and Loveland [RL78] show that for formulas of length n with $m > 0$ quantifier alternations, complexity is just $2^{2^{cn^{m+4}}}$ for constant $c > 0$. More specifically, Haase [Haa14] shows that Presburger Arithmetic formulas with fixed quantifier alternations are *complete* for respective levels of the weak EXP hierarchy. Woods [Woo15] shows that sets described by Presburger formulas are exactly those sets which have rational generating functions.

Cooper's algorithm is by no means the only approach to checking validity of Presburger Arithmetic formulas. Presburger Arithmetic formulas with fewer than two quantifier alternations are already similar to integer linear programming problems, which are NP-hard, and NP, for one or no quantifier alternations respectively. For quantifier-free problems, the Boolean structure of the formula has a large effect on performance. This can be addressed by specialized techniques that use SAT solvers to break the formula down into conjuncts. These techniques include projection [Mon10] and abstraction [KOSS04]. Additionally, the *Omega Test* described by Pugh [Pug91] can be used for efficient solving of quantifier-free Presburger Arithmetic formulas. Yet further afield, Boudet and Comon [BC96] give an automata-based method for solving Presburger formulas, and give tight performance bounds (including for formulas with no quantifier alternations). Certain applications have been described which take advantage of automata-based methods [SKR98, CJ98].

The combination of Presburger Arithmetic with various theories has also been investigated, and some combinations with data structure theories will be mentioned in the next section. Such combinations need to be carefully managed: Downey [Dow72] and, later, Halpern [Hal91] show that adding just one uninterpreted unary predicate to $\Sigma_{\mathbb{Z}}$ is sufficient to make the validity problem Π^1_1-complete.

[1] http://www.isa-afp.org/entries/LinearQuantifierElim.shtml

2.3.2 Theories of Data Structures

2.3.2.1 ARRAY

The theory of *read-over-write arrays* was first given by McCarthy [McC62]. The theory presented here will be parameterized by index and element sorts I and E respectively. The sort of arrays is $ARRAY$, and the signature of array theories is $\Sigma_{ARRAY} = \{\mathsf{read} : ARRAY \times I \to E, \mathsf{write} : ARRAY \times I \times E \to ARRAY\}$. What follows is the extensional theory of arrays $T^{=}_{ARRAY}$, where equality between arrays is defined.

$$(1)\quad \mathsf{read}(\mathsf{write}(a,i,e),i) \approx e \qquad (2)\quad (\forall i.\ \mathsf{read}(a,i) \approx \mathsf{read}(b,i)) \Rightarrow a \approx b$$
$$(3)\quad (i \not\approx j) \Rightarrow \mathsf{read}(\mathsf{write}(a,j,e),i) \approx \mathsf{read}(a,i)$$

Note that the use of monomorphic sorts complicates the use of nested arrays. This is because it is required for the sort $ARRAY$ to be disjoint from the element sort. It is possible to use only a single sort and add a predicate *atomic* which defines a subset of arrays that never contain other arrays, then to add axioms for those atomic arrays that define the element theory, though this method of theory combination is rarely used.

The set of axioms without (2) defines the non-extensional theory of arrays T_{ARRAY}. In that theory it is not possible to conclude from $a \not\approx b$ that arrays a and b differ at some index.

Both the extensional and non-extensional quantifier-free fragments are decidable, although the full theory isn't [BM07]. Armando et al. [ABRS09] show that the Superposition calculus can decide satisfiability in the extensional quantifier-free fragment after removing disequalities. Bradley et al. [BMS06] give a larger fragment of the language of Σ_{ARRAY} called the *array property fragment* which permits guarded universal quantification over array indices (both uninterpreted and in Presburger Arithmetic).

Definition 2.3.1 (Array Property Fragment)**.** Given formulas of the form

$$\forall i : \mathbb{Z}.\ F[i] \Rightarrow G[i] \tag{2.1}$$

where

1. Any occurrence of i in G has the form $\mathsf{read}(a,i)$ for some constant $a : ARRAY$ and such terms never occur below other read operators.

2. F is a $\Sigma_{\mathbb{Z}}$-formula for which

   - the only logical operators are $\wedge$, $\vee$, $\approx$, and
   - the only $\Sigma_{\mathbb{Z}}$ predicate is $\leq$, and
   - universally quantified variables i occur only as immediate subterms of $\leq$ predicates or equations and never in a linear term like $3i + a$ for example.

Then, the array property fragment contains the existential closure of formulas (2.1) and Boolean combinations thereof. Existentially quantified variables are permitted in both F and G.

For the array property fragment T_{ARRAY}-satisfiability is decidable, that is, satisfiability with respect to the *non-extensional* theory of arrays. The decision procedure for this fragment rewrites universal quantifiers by instantiating them over a finite set of relevant indices (consisting of any existentially quantified Presburger terms in the guard formulas as well as any other $\mathbb{Z}$-sorted constants present), followed by the application of a decision procedure for the quantifier-free fragment.

Ghilardi et al. [GNRZ07] extend Σ_{ARRAY} with extra functions, while preserving decidability in the quantifier-free fragment. In particular, a *dimension* function is introduced which returns the largest initialized index, i. e. , the size of the array. As this is a function from $ARRAY$ to $S_{\mathbb{Z}}$, the decidability of the satisfiability problem of the extended theory does not immediately follow from classical theory combination results.

Kapur and Zarba [KZ05] reduce decidability of the quantifier-free fragment of T_{ARRAY} to a simpler theory of uninterpreted functions with equality. Such a reduction is necessarily exponential, as T_{ARRAY}-satisfiability is NP-complete, while satisfiability in the quantifier-free uninterpreted fragment is $O(n \log n)$. *Combinatory Array Logic* [dMB09] (also a fragment of the theory of uninterpreted functions) includes the language of Σ_{ARRAY}, and admits a decision procedure for satisfiability in the ground conjunctive fragment.

Ge and de Moura [GdM09] give some quantified fragments for which the satisfiability problem is decidable, also using finite quantifier instantiation. One of these fragments properly generalizes the Array Property fragment above. Critically, this paper describes a method for finding the set of instances required to instantiate the universal quantifiers.

Ihlemann et al. [IJSS08] describe *local theories* which are equisatisfiable to some finite set of ground instances. Local theory extensions are extensions of some base theory with a local theory, such that the extension part can be reduced again to a finite set of ground instances. It is shown that the Array Property fragment and several others are in fact local theory extensions.

Armando et al. [ABRS09] give a method for deciding quantifier-free array formulas with the Superposition calculus, in which a critical part is the elimination of atoms involving the extensionality axiom.

2.3.2.2 LIST

This is the theory of LISP style lists over element sort E, with signature $\Sigma_{LIST} = \{\mathsf{nil} : LIST, \mathsf{head} : LIST \to E, \mathsf{tail} : LIST \to LIST, \mathsf{cons} : E \times LIST \to LIST\}$.

(1) $x \approx \mathsf{nil} \vee (\mathsf{cons}(\mathsf{head}(x), \mathsf{tail}(x))) \approx x$
(2) $\mathsf{cons}(x, y) \not\approx \mathsf{nil}$
(3) $\mathsf{head}(\mathsf{cons}(x, y)) \approx x$
(4) $\mathsf{tail}(\mathsf{cons}(x, y)) \approx y$

T_{LIST} has a sub-theory T^A_{LIST} of *acyclic* lists– lists which do not contain copies of themselves at any depth. This sub-theory also satisfies the list axioms, however, there is no finite set of axioms which can differentiate the general list theory from T^A_{LIST}. In general, reasoning in T^A_{LIST} is simpler than in T_{LIST}, hence decision procedures operate w. r. t. the former theory. When reasoning using the axioms above, results hold in the general theory T_{LIST}.

Oppen [Opp80] shows that T_{LIST}-satisfiability problem for the quantifier-free acyclic fragment is linear in the number of literals, while validity in the full first-order theory is decidable but non-elementary.[2] Zhang et al. [ZSM04] give results for quantifier-free formulas of lists with a length function (defined in Presburger Arithmetic). Decidability of this combination is not immediate from the Nelson-Oppen combination theorem, as that requires theory signatures to be disjoint.

As above, Kapur and Zarba [KZ05] describe a reduction from the theory of lists to a simpler sub-theory of constructors only. Suter et al. [SDK10], inspired by functional programming techniques, use homomorphisms on the term algebra of various theories to reduce decidability problems in one theory to another. This reduction is contingent on the property to be proved, for example, lists may be reduced to sets when containment is in question. Also by Suter et al. [SKK11a] is a theorem proving method that reasons 'modulo recursive theories' by taking an iterative deepening approach: interleaving model finding and expansion of recursive function definitions. This naturally applies to the theory of lists and functions defined over lists.

2.3.2.3 Recursive Data Structures

The theory of recursive data structures T_{RDS} is a natural generalization of the $LIST$ theory, and many of the results about T_{LIST} apply to it. It has signature $\Sigma_{RDS} = \{c : S_1 \times \ldots \times S_n \to R_c, p_1 : R_c \to S_1, \ldots, p_n : R_c \to S_n, \mathsf{atom} : R_c \to \mathit{Bool}\}$, where R_c is the sort of data structures constructed by c.

$$(1)\quad \mathsf{atom}(x) \vee c(p_1(x), \ldots, p_n(x)) \approx x \qquad (2)\quad \neg\mathsf{atom}(c(x_1, \ldots, x_n))$$
$$(3.1)\quad p_1(c(x_1, \ldots, x_n)) \approx x_1 \ \ldots \qquad (3.n)\quad p_n(c(x_1, \ldots, x_n)) \approx x_n$$

The symbol c is an n-ary constructor for the structure, and p_i is a projection function on the constructor tuple. Many common data structures are described by this theory, in addition to lists, such as records, binary trees and rose-trees. The type of structure is determined by selecting both the number and sort of constructor arguments. As for lists, there is an acyclic sub-theory of T_{RDS} in which no constructor term can contain itself at any depth.

Typically, decision procedures for T_{LIST} are implemented as decision procedures for T_{RDS}. Other approaches reduce Σ_{RDS}-formulas to set, multiset, or list theories, depending on the conjecture to be checked [SDK10]. Sofronie-Stokkermans [SS05] shows that the theory of recursive data structures is covered by the local fragment.

[2] Although lists can be encoded as integers, this brings no advantage since cons is a pairing function [Opp80] and pairing functions are, at best quadratic polynomials. Therefore, they are not definable in the language of Presburger arithmetic.

However, exhaustive instantiation of the axioms is less efficient than the $O(n \log n)$ decision procedure for acyclic data structures given by Oppen [Opp80].

2.3.3 Local Theories

Local theories are a semantically defined class of theories for which the satisfiability of quantifier-free formulas is equivalent to the satisfiability of a ground instantiation of the theory axioms with terms occurring in the quantifier-free formula.

Definition 2.3.2 (Local Theory)**.** A *local theory* is a set of Horn[3] Σ-clauses H such that, given a ground Horn Σ-clause C, $H \wedge C$ is satisfiable if and only if $H[C] \wedge C$ is satisfiable. $H[C]$ is the set of ground instances of H in which all terms are subterms of ground terms in either H or C.

Local theories can be generalized to the case of hierarchic theories, where a base theory is extended with new operators and axioms which obey certain restrictions. Let T_0 be defined by a (possibly infinite) set of Σ_0-formulas, and $T_0 \subset T_1$ be a set of Σ_1-formulas where $\Sigma_0 \subseteq \Sigma_1$. A *partial interpretation* is a Σ_1-interpretation in which some operators (except those in Σ_0) may be assigned partial functions. A ground term is *undefined* w. r. t. a partial interpretation if its argument lies outside of the domain of its assigned function, or if any of its arguments are undefined. Partial interpretations can model clause sets, with an appropriately modified definition of satisfiability: a *weak partial model* of a set of ground clauses is a partial interpretation such that every clause has either a satisfied literal (in the usual sense of satisfaction) or at least one literal which contains an unknown subterm. Non-ground clause sets are satisfied (weakly) if all of their ground instances are satisfied as above.

Definition 2.3.3 (Local Theory Extension)**.** Let $T_1 = T_0 \cup K$, where K is a a set of clauses defining the extension to theory T_0. For every set G of ground Σ_1-clauses $T_1 \cup G \models \bot$ iff $T_0 \cup K[G] \cup G$ has no weak partial model in which all terms among the ground instances of K and G are defined.

These results and other refinements of locality in hierarchic theorem proving are given in Sofronie-Stokkermans [SS05].

Theories in the local fragment include lists, arrays, and other data structures, as well as monotone functions and free functions over certain base theories. Further results show how to combine local fragments [ISS10], and also describe methods for proving the locality of a clause set using saturation theorem proving techniques [HSS13].

2.4 Saturation Based Proof Calculi

This section gives basic definitions for proof calculi used throughout later sections, as well as common operations on terms and clauses that will be used to describe

[3] universally quantified disjunctions with at most one positive literal

applications of the proof calculi. Definitions will follow Baader and Nipkow [BN98] for term definitions, and Nieuwenhuis and Rubio [NR01] for calculi definitions.

Superposition is a calculus for equational reasoning in first-order clausal logic. This calculus will be assumed as the basis for the reasoning methods described later. It developed from the Paramodulation calculus, which was a version of the classical Resolution calculus for first-order logic extended with the Lebniz rule for equality (i. e. , 'like-replaces-like'). A major advance was the removal of the dependence on axioms for reflexivity by Brand [Bra75]. The Knuth-Bendix completion algorithm [KB83] suggested the use of a term order to restrict the orientation of equations and allowed eliminating inferences below variables [BG94]. Though the main calculus used here is based on the Superposition calculus, these definitions are common to other saturation based calculi (mainly precursors of the Superposition calculus): Resolution calculi and Paramodulation calculi.

The main data structure used by first-order theorem provers is the *clause*: a universally quantified disjunction of possibly negated atomic formulas. The equisatisfiability of general first-order formulas to conjunctions of clauses (i. e. , to clause normal form) is well-known. Due to the associativity and commutativity of disjunction, clauses are usually considered to be multisets of literals and the universal quantifier prefix is left implicit. The empty clause is written $\Box$ and is false in all interpretations.

A *substitution* is a map $\sigma : \mathcal{X} \to T(\Sigma, \mathcal{X})$ such that $\sigma(x) \neq x$ for only finitely many variables x. In a many-sorted language substitutions can only map variables to terms of the same sort. The *domain* and *range* of a substitution σ are defined respectively $Dom(\sigma) = \{x \in \mathcal{X} : \sigma(x) \neq x\}$ and $Range(\sigma) = \{\sigma(x) : x \in Dom(\sigma)\}$. Substitutions are often represented as finite lists of bindings from variables to terms, e. g. , $\sigma = [x_1 \mapsto t_1, \ldots x_n \mapsto t_n]$, in that case $\sigma(x_i) = t_i$ for $1 \leq i \leq n$ and $\sigma(x) = x$ otherwise. The identity substitution ϵ is the identity map on $\mathcal{X}$.

A substitution σ has a unique homomorphic extension to terms, clauses and formulas; the application of this to a term is denoted by writing σ in postfix position. The term $t\sigma$ is called an *instance* of t. An instance is *proper* where $t\sigma \neq t$ and *ground* where $t\sigma$ has no free variables. A substitution σ is *renaming* when $Range(\sigma)$ consists only of variables and σ is a bijective map.

A *matching* of s to t is a substitution μ such that $s\mu = t$. A *unifier* of s and t is a substitution σ such that $s\sigma = t\sigma$. Given substitutions σ_1 and σ_2, σ_1 is *more general* than σ_2 if there is a non-renaming substitution σ such that $\sigma_2 = \sigma_1 \cdot \sigma$, (where $\cdot$ is functional composition). Given two terms s, t there always exists an idempotent, (i. e. , $\sigma \cdot \sigma = \sigma$), *most general unifier*, denoted $\mathrm{mgu}(s, t)$. This is unique up to renaming of variables.

Specific subterms are identified by square brackets, i. e. , $t[s]$ indicates that t has a proper subterm s at *some* position. The outer term is the *context*, formulas and clauses may also be contexts. Where the same context is used twice with different subterms, it indicates a single replacement at the position of the first subterm. If all terms are replaced, both will be written in the brackets: $t[r\backslash s]$ is the result of replacing r everywhere by s.

Superposition calculi are parameterized by a well-founded order $\prec$ on $T(\Sigma, \mathcal{X})$.

This *term ordering* must be closed under substitution $s \prec t \Rightarrow s\sigma \prec t\sigma$, and closed under contexts $s \prec t \Rightarrow r[s] \prec r[t]$. An order that satisfies these properties is called a *reduction order*. All Superposition calculi require the term order to be a reduction order that is total on ground terms.

Typically, this order is implemented as either a Knuth-Bendix or Lexicographic order, both parameterized by a total precedence on symbols of Σ [Der82]. Knuth-Bendix orders also assign a weight to each symbol in Σ, which is factored into the ordering.

The term order $\prec$ is extended to an order on equations, literals and clauses using multiple applications of the multiset extension [BN98]. Specifically, equations $l \approx r$ become the multiset $\{l, r\}$, negated equations become $\{l, l, r, r\}$ (by convention, negated equations sort higher than their positive forms), and clauses $L_1 \vee L_2 \vee \ldots$ become $\{L_1, L_2, \ldots\}$. An order on multisets $\prec_m$ can be constructed from an existing order $\prec$ (namely, the term order) by setting $S_1 \prec_m S_2$ if and only if $S_1 \neq S_2$, and if there are more of some e in S_1 than S_2, then there is a larger (relative to $\prec$) e' for which there is more of e' in S_2 than in S_1. If $\prec$ is well-founded and total, then so is $\prec_m$.

A *proof calculus* consists of *rules* that describe a map from sets of premise clauses to sets of conclusion clauses.

Definition 2.4.1 (Calculus Rule)**.** An rule

$$\frac{P}{R} \text{ if } Cond$$

consists of multisets P and R of schematic clauses[4], the *premises* and conclusions respectively. *Cond* is an optional condition that restricts which clauses satisfy the schematic clauses in P and R.

There are two types of rules, differentiated by their action on a clause set: *inference rules* which only introduce a single clause, and *simplification rules* which may remove or alter clauses in a clause set.

Definition 2.4.2 (Application of Calculus Rules)**.** For an inference rule with premises P, conclusion $\{C\}$ and condition *Cond*, the application of the rule to clause set $\mathcal{N}$ is possible iff $P' \subseteq \mathcal{N}$ is an instance of the clause schema P that satisfies *Cond*, and the result is $\mathcal{N} \cup \{C'\}$, where C' is the corresponding instance of schema C.

For a simplification rule with conclusion R, assuming $P' \subseteq \mathcal{N}$ satisfies *Cond* as for inference rules, the result is $\mathcal{N} \setminus P \cup R'$, with R' an instance of schema R.

A calculus is *sound* w. r. t. the usual logical consequence relation $\models$, if for any rule with premises P and conclusion set R, for each $C \in R$, $P \models C$. A calculus is *refutation*

[4] A clause whose variables range over arbitrary terms, literals and clauses. They are common in the literature, and their function will be clear when concrete inference rules are given, so a full definition is omitted.

complete, if any clause set that is closed w. r. t. the calculus rules and that does not contain $\square$ is satisfiable.

A *derivation* is a sequence of clause sets $\mathcal{N}_0, \mathcal{N}_1, \ldots$ such that $\mathcal{N}_{i+1}$ is the result of the application of some rule to $\mathcal{N}_i$. Where both inference and simplification rules are used, a notion of *redundancy* is needed to ensure that looping behaviour is avoided. A ground clause C is made *redundant* by a set of ground clauses $\mathcal{N}$ when for $\{C_0, \ldots, C_n\} \subseteq \mathcal{N}$ such that $C_i \prec C$, $\{C_0, \ldots, C_n\} \models C$. Then, a non-ground clause is redundant w. r. t. clause set $\mathcal{N}$ if the set of ground instances of C is redundant w. r. t. the ground instances of $\mathcal{N}$ and if $C \in \mathcal{N}$, then C is redundant w. r. t. $\mathcal{N}$. An inference is redundant if the conclusion is redundant w. r. t. clauses smaller than the maximal premise.

Then, redundant inferences need not be performed, and redundant clauses can safely be deleted by simplification rules.

Given a derivation $\mathcal{N}_0, \mathcal{N}_1, \ldots$ the set of *persistent* clauses (also called the *saturation* of a clause set w. r. t. a calculus) is defined as $\mathcal{N}_\infty = \bigcup_{0 \leq i} \bigcap_{i \leq j} \mathcal{N}_j$. Lastly, there is a restriction on the order of inferences: a derivation $\mathcal{N}_0, \mathcal{N}_1, \ldots$ is *fair* with respect to a set of inference and simplification rules $CALC$, if for every inference π of I with premises in $\mathcal{N}_\infty$ there is a $j \geq 0$ for which π is redundant with respect to $\mathcal{N}_j$. In other words, no necessary inference is postponed indefinitely.

Definition 2.4.3 (Refutation Complete)**.** A calculus is *refutation complete* iff from any unsatisfiable clause set $\mathcal{N}$ every fair derivation contains $\square$.

2.5 Superposition for Hierarchic Theories

Superposition for hierarchic theories (briefly: Hierarchic Superposition) [BGW94, BW13b], is a modification of the standard Superposition calculus for reasoning in a hierarchic combination of first-order equational logic and some interpreted theory.

A *specification* consists of a signature Σ and a set $\mathcal{B}$ of Σ-interpretations that is closed under isomorphism; called the *base or background theory*. A *hierarchic specification* $(\Sigma, (\Sigma_\mathrm{B}, \mathcal{B}))$ has a *base* (or background) specification $(\Sigma_\mathrm{B}, \mathcal{B})$, and an extended signature $\Sigma \supset \Sigma_\mathrm{B}$.

In this section it is assumed that interpretations in the specification are *term-generated*, i. e. , all members of the domain of an interpretation are the image of some term of the language. By the Löwenheim-Skolem theorem, term-generated interpretations are sufficient to model any infinite first-order theory, so long as the language contains a countable infinity of terms. If it does not (e. g. , it has no non-constant function symbols), an infinite set of constants can be added to the signature.

For a given background specification $\mathcal{B}$, $\mathrm{GndTh}(\mathcal{B})$ is the set of all ground formulas in the language of Σ_B satisfied by all interpretations in $\mathcal{B}$.

The most common base specification will be LIA with parameters as defined in Section 2.3.1, however, any decidable theory can be used (this could be weakened to semi-decidable theory, as the overall goal is just refutation completeness). A feature of theories that admit quantifier elimination is that their joint theory also

admits quantifier elimination, and for that reason the background theory is viewed as indivisible. The background theory need not be arithmetic or numerical either. A formula is satisfiable w. r. t. a specification $(\Sigma, (\Sigma_B, \mathcal{B}))$ if it is satisfied by a Σ-interpretation whose Σ_B-reduct is in $\mathcal{B}$.

Example 2.5.1. A first-order logic formula in clause normal form is in the *effectively propositional fragment* (EPR) iff its literals are composed of only predicates, variables, and constant symbols. The validity problem for the effectively propositional fragment is NEXP-time complete. Consider a hierarchic specification $(\Sigma, (\Sigma_B, \mathcal{B}))$, where $\mathcal{B}$ is the set of all Σ_B-interpretations, Σ_B has only predicates, and there are no functions in Σ whose result sort is a base sort. Then Σ_B-clauses are in the EPR fragment, and, by results later in this section, the combination of the Superposition calculus with a solver complete on the EPR fragment (e. g. , iProver or Darwin) is refutation complete.

Less obvious is the fact that EPR solvers can also be used when the sorts of Σ_B and Σ are non-cyclic (i. e. , for every sort S in Σ, there are no S sorted terms which have S sorted subterms), see Korovin [Kor13]. That generalizes EPR in the sense that functions between sorts of Σ_B are permitted, so long as they meet the non-cyclic condition.

Certain base specifications have a distinguished set of constants called *domain elements*. These are abstractly characterized as the largest set of ground Σ_B-terms that are pairwise distinct in all models of the background specification, and minimal w. r. t. the term order. The set of domain elements for a particular specification is written $Dom(\Sigma_B)$, e. g. , $Dom(\Sigma_{\mathbb{Z}}) = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

The calculus requires that operators in Σ_B have a lower precedence than any in $\Sigma \setminus \Sigma_B$.

In Bachmair et al. [BGW94] substitutions, in particular unifiers and instantiations, were restricted by only allowing Σ_B-terms to be substituted for background sorted variables. By restricting substitutions, the number of possible inferences is greatly reduced and thus prover efficiency should increase. In Baumgartner and Waldmann [BW13b] this restriction was made sharper by restricting substitutions so that for a subset of background sorted variables only domain elements could be substituted in. These variables are known as *abstraction* variables, any other variables are *general* variables. The set of variables is divided: $\mathcal{X} = \mathcal{X}^A \cup \mathcal{X}^G$, where $\mathcal{X}^A$ are the abstraction variables. For this section only, abstraction variables will be written capitalized, while general variables will be lower-case. In other sections the distinction is usually unnecessary.

Any term in $T(\Sigma_B, \mathcal{X}^A)$ is a *pure* term. A substitution σ where every $X \in \mathcal{X}^A$ $\sigma(X)$ is pure is a *simple* substitution. An important consequence is that only pure terms can be substituted into pure terms. A *simple instance* of a term or clause is formed by the application of a simple substitution. The set of *simple ground instances* of a term (clause) t is denoted $\mathrm{sgi}(t)$. The set $\mathrm{sgi}(X)$, for abstraction variable X consists of just domain elements or ground Σ_B-terms which, by definition, are always equivalent to a domain element. Then abstraction variables can be seen as placeholders for

domain elements. For non-ground terms s, t $s \prec t$ only if for all instances s', t' of the respective terms $s' \prec t'$. Since the only possible instances of abstraction variables are domain elements, for any non-pure term t and any abstraction variable X, it follows that $X \prec t$.

The set of simple ground instances is critically important for understanding the completeness result of the Hierarchic Superposition calculus. Essentially, the calculus works by simulating, in a certain technical sense, a derivation from the ground instances of a clause set and the (typically infinitely many) ground theorems of the base specification $\mathrm{GndTh}(\mathcal{B})$. For efficiency, the simulation uses simple ground instances only then, for every model M of the full set of ground instances of clauses $\mathcal{C}$ it is required that $M, \mathcal{B} \models \mathcal{C} \Leftrightarrow \mathrm{sgi}(\mathcal{C})$. Furthermore, any model of the simple ground instances when reduced to the signature Σ_B must be in the base specification model class. Specifically, the cardinalities of the carrier sets for the base sorts must agree with an actual model of the base specification. This can be broken into two considerations: *no confusion* and *no junk*; confusion is where elements of a base sort are incorrectly equated, junk refers to extra elements included in the carrier set of an interpretation that do not appear in a base interpretation.

Although confusion is prevented by including $\mathrm{GndTh}(\mathcal{B})$, (implemented as a check to a theory solver) junk elements may appear when they are never identified as some member of the base sort in the course of a derivation. Thus, preventing junk (as a prerequisite for refutation completeness) requires an extra assumption on clause sets– the *sufficient completeness* property.

Definition 2.5.1 (Sufficient Completeness w. r. t. Simple Ground Instances)**.** A clause set $\mathcal{N}$ has sufficient completeness (w. r. t. simple ground instances) iff for *every* first-order model M (not necessarily extending the base specification $\mathcal{B}$) of $\mathrm{sgi}(\mathcal{N}) \cup \mathrm{GndTh}(\mathcal{B})$ and any base sorted ground term t in $\mathrm{sgi}(\mathcal{N})$, $M \models t \approx e$ for some ground Σ_B-term e.

In the following, 'sufficient completeness w. r. t. simple ground instances' will be abbreviated to just 'sufficient completeness'. This property is undecidable for general clause sets, as it can be reduced to the non-ground rewrite rule termination problem. It is also loosely connected with the idea of first-order definability: any clause set in which all the base-sorted free operator symbols are defined w. r. t. Σ_B will have sufficient completeness, although the reverse does not hold. Completeness will be discussed further in Section 2.5.3.

2.5.1 Calculus Rules

The Hierarchic Superposition calculus consists of the rules Equality Resolution, Superposition, Negative Superposition, Factoring, Close described below. The original calculus had a version including equality factoring and merging paramodulation, however those rules do not work with weak abstraction [BW13b].

$$\text{Equality Resolution}\ \frac{l \not\approx r \vee C}{C\sigma}$$

if (i) neither l nor r is a pure Σ_B-term, (ii) $\sigma = \mathrm{mgu}(l, r)$ is simple, (iii) $(l \not\approx r)\sigma$ is maximal[5] in $C\sigma$

$$\text{Superposition}\ \frac{l \approx r \vee C \qquad s[l'] \approx t \vee D}{(s[r] \approx t \vee C \vee D)\sigma}$$

if (i) neither l nor l' is a pure Σ_B-term, (ii) l' is not a variable, (iii) $\sigma = \mathrm{mgu}(l, l')$ is simple, (iv) $l\sigma \not\preceq r\sigma$, (v) $(l \approx r)\sigma$ is strictly maximal in $C\sigma$, (vi) $s\sigma \not\preceq t\sigma$, (vii) $(s \approx t)\sigma$ is strictly maximal in $D\sigma$

$$\text{Negative Superposition}\ \frac{l \approx r \vee C \qquad s[l'] \not\approx t \vee D}{(s[r] \approx t \vee C \vee D)\sigma}$$

if (i) neither l nor l' is a pure Σ_B-term, (ii) l' is not a variable, (iii) $\sigma = \mathrm{mgu}(l, l')$ is simple, (iv) $l\sigma \not\preceq r\sigma$, (v) $(l \approx r)\sigma$ is strictly maximal in $C\sigma$, (vi) the first premise does not have selected literals, (vii) $s\sigma \not\preceq t\sigma$, and (viii) $(s \not\approx t)\sigma$ is maximal in $D\sigma$.

$$\text{Factoring}\ \frac{l \approx r \vee s \approx t \vee C}{(r \not\approx t \vee l \approx t \vee C)\sigma}$$

if (i) neither l nor s are pure Σ_B-terms, (ii) $\sigma = \mathrm{mgu}(l, s)$ is simple, (iv) $l\sigma \not\preceq r\sigma$, (iv) $s\sigma \not\preceq t\sigma$, (v) $(s \approx t)\sigma$ is maximal in $(l \approx r \vee C)\sigma$.

$$\text{Close}\ \frac{C_1 \quad \cdots \quad C_n}{\square}$$

if $C_1, \ldots, C_n$ are Σ_B-clauses and $\{C_1, \ldots, C_n\}$ is unsatisfiable w. r. t. $(\Sigma_B, \mathcal{B})$.

The main differences between these rules and those of the standard Superposition calculus[BG94]: the use of simple unifiers; the requirement that the literals selected for inferencing are never pure terms from the base signature; and the fact that abstraction must be performed after every inference (omitted in the rules).

The original Hierarchic Superposition calculus included an undecidable maximality condition based on the existence of a simple grounding substitution. Specifically, each rule except constraint refutation requires that there be a simple substitution ψ such that $(u \approx v)\sigma\psi$ is a maximal occurrence of an equation in the relevant ground instance of the premise (e. g., $(C \vee u \not\approx v)\sigma\psi$ for Equality Resolution). The authors note that it can be replaced by '$(u \approx v)\sigma$ is maximal and non-base', but that this is weaker, because $f(x) \approx f(x)$ and $f(t) \approx f(t)$ are incomparable when x is not a subterm of t for example, meaning two applications of the rule will obtain. However, $f(x) \approx f(x)$ is smaller when substitutions are restricted to domain elements. This

[5] i. e., maximal in the set of literals making up the clause $C\sigma$

case is partly covered by the introduction of abstraction variables, and by ordering abstraction variables strictly less than any impure terms.

The notion of redundancy in Hierarchic Superposition is only slightly modified from the definition of redundancy for the regular Superposition calculus. A clause C is redundant w. r. t. a set of clauses $\mathcal{N}$ if each clause in $\text{sgi}(C)$ is redundant w. r. t. $(\text{sgi}(\mathcal{N}) \cup \text{GndTh}(\mathcal{B}))$ under the usual Superposition calculus (i. e. , is entailed by smaller clauses). Similarly, inferences from $\mathcal{N}$ are redundant if all simple ground instances of the conclusion are redundant w. r. t. $(\text{sgi}(\mathcal{N}) \cup \text{GndTh}(\mathcal{B}))$.

The inference system consisting of the above rules and a non-deterministic simplification rule which removes any clause as long as it is redundant, is referred to as *HSP*. Soundness of *HSP* follows from soundness of each of the rules:

Theorem 2.5.1 ([BGW94, BW13b])**.** *If the set of persisting clauses $\mathcal{N}_\infty$ in a $SUP_\prec$ derivation from clause set $\mathcal{N}$ does not contain $\square$, then $\mathcal{N}$ is satisfiable.*

2.5.2 Abstraction

Abstraction is a technique for transforming a clause which contains literals over the mixed signature $\Sigma = \Sigma_B \cup \Sigma_F$ into an equivalent clause in which each literal is over either just Σ_B or just Σ_F. This is done by the following transform on a clause C: Where t is a Σ_B-term and $f \notin \Sigma_B$:

$$\text{replace } C[f(\ldots, t, \ldots)] \text{ with } C[f(\ldots, x, \ldots)] \vee x \not\approx t$$

and similarly, where $f \in \Sigma_B$ and t is a Σ_F-term. This is repeated until all literals are either over Σ_B or over Σ_F. For clause C, the limit of this process is denoted $\text{abstr}(C)$ and the result is called the *(fully) abstracted form* of C.

An advantage of full abstraction is that conclusions of inferences in the Superposition calculus never need abstraction when the premises are abstracted. Abstraction also allows for a limited form of theory unification below FG operators, i. e. , given terms s, t find substitution σ such that $T \models s\sigma \approx t\sigma$ for theory T.

Example 2.5.2. Let $C = f(g(c) + 10) \not\approx f(20)$, then

$$\text{abstr}(C) = f(w_1) \not\approx f(w_2) \vee w_1 \not\approx w_3 + 10 \vee w_2 \not\approx 20 \vee w_3 \not\approx g(c)$$

The conclusion of an equality resolution inference with $\sigma = [w_1 \to w_2]$:

$$w_2 \not\approx w_3 + 10 \vee w_2 \not\approx 20 \vee w_3 \not\approx g(c)$$

Abstraction greatly increases the number of possible inferences, because it removes structure inside terms. In the previous example, abstraction of the subterm $f(20)$ to $f(w_2)$ means that paramodulation with, e. g. , $f(w) \approx 0 \vee w \not\approx 10$ is possible. While abstraction allows some necessary inferences, it also allows many spurious inferences such as the above[6], not all of which can be shown to be trivial as easily.

[6]This fact, and difficulties guaranteeing completeness are cited as the reason for the decoupled

Then, any restriction of abstraction which does not affect the completeness of the calculus would bring performance improvements.

The original presentation of the Hierarchic Superposition calculus [BGW94] used full abstraction, which actually makes the calculus incomplete on certain clause sets [BW13b].

Example 2.5.3 (Full Abstraction Destroys Sufficient Completeness)**.** Consider the clause set $\{f(1) < f(1), \neg(f(1) < f(1))\}$. Since it has no first-order models it has sufficient completeness (trivially). The fully abstracted form is: $\{x_1 < x_2 \vee x_1 \not\approx f(1) \vee x_2 \not\approx f(1), \neg(y_1 < y_2) \vee y_1 \not\approx f(1) \vee y_2 \not\approx f(1))\}$. No inference rules in *HSP* apply, as the complementary literals $(x_1 < x_2), \neg(y_1 < y_2)$ are base and all non-base literals are negative, so *HSP* is unable to produce a refutation of the abstracted clause set.

Instead, Baumgartner and Waldmann [BW13b] propose two solutions: *weak abstraction*, which limits abstraction to only necessary subterms while preserving sufficient completeness, and *abstraction variables*, which only stand for domain elements. Using abstraction variables limits the number of instances a clause can produce, while allowing more terms to be ordered (since an abstraction variable sorts lower than any non-background or non-pure term).

Definition 2.5.2 (Weak Abstraction [BW13b])**.** A term $t \in T(\Sigma_B, \mathcal{X})$ that is neither a variable nor a domain element, is a *target term* in clause C if t occurs in a subterm of C having the form:

1. $f(\ldots, t, \ldots)$, for $f \in \Sigma \setminus \Sigma_B$.
2. $g(\ldots, t, \ldots, s)$, for $g \in \Sigma_B$ and $s \notin T(\Sigma_B, \mathcal{X})$.

As in full abstraction, target terms t are abstracted out using fresh abstraction variables: $C[t]$ becomes $C[X] \vee X \not\approx t$ if t is pure or $C[x] \vee x \not\approx t$ otherwise. The *weak abstraction* of C, weak(C) is the limit of this process; weak(C) is equivalent to C and contains no target terms.

Example 2.5.4. Continuing Example 2.5.2, weak$(C) = f(g(c) + 10) \not\approx f(20)$ and so no equality resolution inference is possible. Now consider the unsatisfiable clause set $\{C, g(c) \approx 10\}$. A derivation beginning from $\{\text{abstr}(C), \text{abstr}(g(c) \approx 10)\}$ yields the $T_{\mathbb{Z}}$-unsatisfiable clause

$$w_2 \not\approx w_3 + 10 \vee w_2 \not\approx 20 \vee w_3 \not\approx 10$$

by paramodulation into the final clause of Example 2.5.2.

approach taken by SPASS+T [WP06]

A derivation from the weakly abstracted clause set $\{C, g(c) \approx 10\}$:

$f(10+10) \not\approx f(20)$	by paramodulation
$f(w_1) \not\approx f(20) \vee w_1 \not\approx 10+10$	by weak abstraction
$20 \not\approx 10+10$	by equality resolution
$\square$	constraint refutation

Note that the first clause in the derivation was not weakly abstracted; unlike full abstraction, weak abstraction is not preserved by superposition inferences in general.

Though the same inference steps were performed in both cases, the clauses in the latter derivation were much smaller. This is advantageous given the super-exponential time complexity of Cooper's algorithm.

Apart from the performance improvements illustrated above, the ultimate aim of weak abstraction was to avoid incompleteness caused by the abstraction procedure:

Theorem 2.5.2 (Prop. 5.2 of [BW13b])**.** *Clause set $\mathcal{N}$ has sufficient completeness if and only if* weak($\mathcal{N}$) *does.*

This follows because sgi($\mathcal{N}$) and sgi(weak($\mathcal{N}$)) have the same first-order models.

Example 2.5.5. Let $\mathcal{N} = \{f(1) < f(1), \neg(f(1) < f(1))\}$. Then weak($\mathcal{N}$) $= \mathcal{N}$ as 1 cannot be abstracted being a domain element, and $f(1)$ cannot be abstracted as it is not in $T(\Sigma_B, \mathcal{X})$.

Theorem 2.5.3 ([BW13b])**.** *Let $I \in \mathcal{B}$ be a term-generated Σ_B-interpretation and let $\mathcal{N}$ be a set of weakly abstracted Σ-clauses. If I satisfies all Σ_B-clauses in* sgi($\mathcal{N}$) *and $\mathcal{N}$ is saturated w. r. t. HSP, then $\mathcal{N}_I$ (i. e. . $\mathcal{N}$ reduced by I) is saturated with respect to the standard Superposition calculus.*

2.5.3 Completeness

Bachmair et al. [BGW94] give two requirements for the refutational completeness of Hierarchic Superposition. The first is sufficient completeness, as defined above. The second requirement is *compactness* of the base specification. A specification is called *compact*, if every set of formulas that is unsatisfiable w. r. t. the specification has a finite unsatisfiable subset. This is required because only *finite* cardinality clause sets can be passed to a reasoner for the base specification.

Note that sufficient completeness must be proven for each input clause set, while compactness is a general property of the base specification. Further, if a base specification is not compact, the consequence is that certain proofs will not terminate. For example, the LIA specification with parameters is not compact as the set of unit clauses $\{(0 < \alpha), \ldots, (n < \alpha), \ldots\}$ is unsatisfiable w. r. t. the specification, but every subset is satisfiable.[7] A derivation for which the set of persisting clauses includes

[7]The specification for LIA without parameters is compact, however parameters are necessary to recover sufficient completeness later.

the set above cannot be finite in length, as Close will never apply. There are language fragments on which Hierarchic Superposition is refutation complete, but whose specifications are not compact.

Theorem 2.5.4 (Complete but Non-Compact fragment [BW13a])**.** *The Hierarchic Superposition calculus is refutationally complete w. r. t. $T_{\mathbb{Z}}$ for finite sets of Σ-clauses in which every $\mathbb{Z}$-sorted term is either (i) ground, or (ii) a variable, or (iii) a sum $x + k$ of a variable x and a number $k \geq 0$ that occurs on the right-hand side of a positive literal $s < x + k$.*

On the other hand, a derivation from a clause set without sufficient completeness may terminate without deriving an empty clause, yet $\mathcal{B}$-satisfiability cannot be concluded.

Example 2.5.6. Consider the set of unit clauses

$$x < g(x) \qquad g(y) < 100$$

The clause set is unchanged by weak abstraction as all base terms are variables or domain elements. No default Hierarchic Superposition rules apply[8] so the clause set is saturated. Clearly it is not $T_{\mathbb{Z}}$-satisfiable, but it also does not have sufficient completeness as g can be arbitrarily interpreted in models of $\mathrm{sgi}(x < g(x), g(y) < 100) \cup \mathrm{GndTh}(T_{\mathbb{Z}})$. So the calculus is excused from finding a refutation in this case.

Sufficient completeness cannot be decided for general clause sets.[9] For certain classes of Σ-clause sets it is possible to establish a variant of sufficient completeness automatically [KW12, BW13b]. Essentially, if all base sorted non-base terms in the input are ground, it suffices to show that every such term *in the input* is equal to some Σ_B-term. This can be achieved automatically by adding a *definition* $t \approx \alpha_t$ for every base sorted non-base term t occurring in a clause $C[t]$, where α_t is a new parameter (base sorted constant); afterwards $C[t]$ can be replaced by $C[\alpha_t]$.

Clauses in which all base-sorted terms are ground are said to be in the *Ground Base-sorted Term (GBT) fragment*.

Theorem 2.5.5 (GBT fragment [BW13b])**.** *Any clause set from the GBT fragment has sufficient completeness.*

2.5.4 Definitions and Sufficient Completeness

The action of adding definitions for terms that possibly break sufficient completeness can be generalized to a calculus rule:

$$\text{Define}\ \frac{C[s]}{s \approx \alpha_s}$$

[8] Unsatisfiability could be concluded by an application of chaining to the LIA inequalities, see Chapter 3

[9] There are decidability results in the literature on algebraic specifications [KNZ87], however, those are usually restricted to positive equations only.

where s is a ground base-sorted term not already defined and α_s is a fresh base-sorted constant.

By construction, $s \approx \alpha_s$ always reduces $C[s]$ to $C[\alpha_s]$, as well as any other occurrences of s in the clause set. So an application of Define is usually combined with an immediate application of rewriting simplification.

For the GBT fragment, the define rule is applied in a pre-processing phase which transforms the clause set to an equivalent one with sufficient completeness. The rule can also be applied eagerly in other cases, as even sufficient completeness of a subset of clauses can result in a successful proof.

Bachmair et al. [BGW94] note that, where all base sorted non-base operators have a finite range, e. g. , a binary valued function into $\mathbb{Z}$:

$$f(x) \approx 0 \vee f(x) \approx 1$$

the clause set will have sufficient completeness. In Chapter 6 a generalization of this is given for operators defined by linear polynomials, or projection functions in non-integer theories. Also in that chapter is a discussion of sufficient completeness of the various theories described in Section 2.3

It was observed [BGW94] that if a relational encoding is used for free base-sorted operators (i. e. , n-ary function symbols are replaced by $(n+1)$-ary predicates and functionality axioms), a clause set without any base sorted non-base terms is produced. However, totality axioms for relational encoded functions

$$\forall x \exists y : \mathrm{B}.\ p_f(x, y)$$

always result in a base-sorted Skolem function after CNF transformation, so these axioms are omitted. As a result, if $\mathcal{N}'$ is a copy of clause set $\mathcal{N}$ in which free base-sorted operators are relational encoded (and their instances appropriately translated), then any model of $\mathcal{N}'$ is possibly only a partial model of $\mathcal{N}$, in the sense that some free base-sorted operators may be interpreted as partial functions.

If it is known a priori that all partial models of $\mathcal{N}'$ can be extended to total models, then satisfiability of $\mathcal{N}$ follows. This criterion (embeddability) was shown to be equal to *locality* of a theory [SS05].

Lemma 2.5.1. *A finite saturation of a clause set which is a relational encoding of a local theory extension (of the base specification) implies satisfiability w. r. t. the base specification.*

However, relational encodings typically degrade the performance of solvers using the Superposition calculus.

Theorem 2.5.6 (Refutational Completeness of *HSP*)**.** *If the base specification $(\Sigma_{\mathrm{B}}, \mathcal{B})$ is compact, then HSP is refutationally complete for Σ-clause sets $\mathcal{N}$ with sufficient completeness.*

Proof. Let N_∞ be the limit of a fair derivation from weak(N), specifically N_∞ is saturated w. r. t. HSP_{Base} and R^H. If $\square \notin N_\infty$, then Close does not apply meaning there are no finite $\mathcal{B}$-unsatisfiable subsets of N_∞. By compactness of $(\Sigma_{\mathrm{B}}, \mathcal{B})$, there is a

Σ_B-interpretation $\mathcal{M}$ that satisfies the Σ_B-clauses in $\mathrm{sgi}(N)$. Then, by Thm. 2.5.3, $(N_\infty)_\mathcal{M}$ reduced by the rewrite system defined by $\mathcal{M}$ is saturated w. r. t. the standard superposition calculus and so there is a Σ-interpretation $\mathcal{M}'$ satisfying $(N_\infty)_\mathcal{M}$. It must be shown that this is $\mathcal{B}$-extending. Since N has sufficient completeness, $\mathrm{sgi}(N)$ is equivalent to the set of ground instances of N in the interpretation $\mathcal{M}'$. By definition of $(N_\infty)_\mathcal{M}$, it follows that $\mathcal{M}'$ satisfies each equation and disequation entailed by $\mathcal{M}$. Further, sufficient completeness implies that $\mathcal{M}$ and the Σ_B-reduct of $\mathcal{M}'$ have identical cardinalities. Thus the Σ_B-reduct of $\mathcal{M}'$ is isomorphic to $\mathcal{M}$, and in the model class $\mathcal{B}$, meaning $\mathcal{M}'$ is $\mathcal{B}$-extending. □

An interesting modification is to make the choice of model $\mathcal{M}'$ externally, by the theory solver for example. Then compactness is not a problem– as there is a single model in the specification, and free base-sorted terms must be equal to Σ_B-terms only in the chosen model (sufficient completeness). If the latter condition is met, then saturation under HSP_{Base} immediately implies $\mathcal{B}$-satisfiability, specifically there is a model extending the chosen one. However, unsatisfiability is contingent on the choice of model; to conclude global unsatisfiability, all possible models in $\mathcal{B}$ must be excluded. This is conceivable where the class $\mathcal{B}$ is finite. This idea will return in later chapters as a basis for a hierarchic satisfiability procedure.

It is possible to prove sufficient completeness using a smaller set than $\mathrm{sgi}(\mathcal{N})$, the following definitions capture that set[10]. A *very-simple ground instance* of a clause C is a ground clause $C\sigma$ such that for all $x \in \mathrm{vars}(C)$, all base sorted subterms of the term $x\sigma$ are pure Σ_B-terms. The set of all very-simple ground instances of a clause (set) C is denoted $\mathrm{vsgi}(C)$. Notice that the essential difference between simple and very-simple instances is that the latter requires base sorted subterms in *all* substituted terms to be Σ_B-terms, rather than only terms directly substituted for base sorted variables. A term t is a *relevant term* for $\mathcal{N}$, iff t is among the free base sorted subterms of $\mathrm{vsgi}(\mathcal{N})$. The set of all relevant terms for $\mathcal{N}$ is $\mathrm{rel}(\mathcal{N})$.

Definition 2.5.3 (Local Sufficient Completeness)**.** $\mathcal{N}$ has *local sufficient completeness* iff for every Σ-model μ of $\mathrm{sgi}(\mathcal{N}) \cup \mathrm{GndTh}(B)$ and every term $s \in \mathrm{rel}(\mathcal{N})$ there is a ground Σ_B-term t such that $\mu \models s \approx t$.

Theorem 2.5.7. *If the base specification $(\Sigma_B, \mathcal{B})$ is compact and if the clause set $\mathcal{N}$ has local sufficient completeness for* $\mathrm{rel}(\mathcal{N})$*, then HSP_{Base} is refutationally complete for* $\mathrm{abstr}(\mathcal{N})$*.* [11]

Proof. (Sketch) Transform $\mathcal{M}'$ from Thm. 2.5.6 into a term-generated Σ-interpretation $\mathsf{nojunk}(\mathcal{M}')$ without extra elements (specifically those not in $Dom(\Sigma_B)$) in base sorts in two steps: In the first step, obtain $\mathcal{M}_0$ from $\mathcal{M}'$ by deleting all additional elements from $S^{\mathcal{M}'}$ where S is a base sort, also redefining $\mathcal{M}'(f)$ arbitrarily whenever $\mathcal{M}'(f(a_1, \ldots, a_n)$ is not in $Dom(\Sigma_B)$. In the second step, take the Σ-interpretation $\mathsf{nojunk}(\mathcal{M}')$ to be the term-generated sub-interpretation of $\mathcal{M}_0$.

[10] Due to Baumgartner and Waldman. Publication pending.
[11] Baumgartner, Waldmann Unpublished draft, 2015

In the last stage of the proof of Thm. 2.5.6, sufficient completeness is used to show that satisfying simple ground instances of $\mathcal{N}$ is equivalent to satisfying normal ground instances. So the Σ_B-reduct of the interpretation is in the base specification.

Then the important properties of $J = \mathsf{nojunk}(\mathcal{M}')$ for the proof are:

1. Every ground instance of a term (or clause) is equal in J to some very-simple instance of the same term (clause), and this preserves truth value of clauses.

2. J and its Σ_B-reduct are term-generated interpretations, and J's Σ_B-reduct satisfies the entailed equations and disequations of the original Σ_B-interpretation $\mathcal{M}$.

3. Very-simple instances of terms and clauses evaluate to the same element under $\mathcal{M}'$ and J.

Item 2) ensures the Σ_B-reduct of J is a member of the base specification and items 1) and 3) show that J satisfies all ground instances of $\mathcal{N}$. □

2.6 Other Reasoners with Interpreted Theories

2.6.1 SUP(LA)

Althaus et al. [AKW09] describe an instantiation of Hierarchic Superposition for the theory of linear rational arithmetic, which is called SUP(LA).

They claim a more modular approach than other applications of hierarchic theorem proving to rationals, such as in Korovin and Voronkov [KV07]. Specifically, Althaus et al. describe techniques for clause simplification specific to reasoning in rational arithmetic (using Farkas' lemma) which enables efficient tautology deletion and subsumption between clauses with rational components.

In contrast to the method described above, SUP(LA) does not allow shared parameters with the base theory, which gives compactness (as there is a single theory in the specification). This is at the cost of restricting the fragment which SUP(LA) can accept, however. The problems due to complete abstraction are also found in this calculus.

The paper also mentions applications to timed automata, some of which can be described using first-order formulas equivalent to a clause set with sufficient completeness. Additionally the authors prove that data structure theories over their background theory are sufficiently complete. A similar result is given in a later chapter.

2.6.2 SMT

The Satisfiability Modulo Theories (SMT) approach has become popular in recent years [BSST09]. It provides a way to both leverage the performance advances found in SAT solvers while providing efficient theory-specific reasoning capability at the same time.

SMT solvers use one of two complementary approaches for theory reasoning: the 'eager approach' in which the input formula is translated to an equisatisfiable propositional formula using relevant theory axioms, then discharged by a SAT solver. Research in this area focuses on the translation step for particular theories and is particularly effective for fixed-width bitvector arithmetic theories. The other main approach– 'lazy encoding', has the SAT solver work on an abstracted 'propositional skeleton' and (one of many) theory solvers testing sets of ground theory literals entailed by the current assignment. The propositional skeleton is made by simply replacing ground theory literals uniformly with new propositional variables. For example a formula $(5 + a \not\approx 7 + y \vee a \not\approx y) \wedge (a \approx x)$ could be abstracted to $(\neg A \vee \neg B) \wedge A$ where A and B are propositional variables. If the propositional skeleton is unsatisfiable, then the formula as a whole is unsatisfiable. On the other hand, the SAT solver could find an assignment to theory literals that satisfies the propositional skeleton– this produces the conjunction of theory literals for the theory solver. If satisfiable, the problem overall is satisfiable w. r. t. the theory, otherwise a different assignment must be found.

The decoupling of solvers is advantageous as a single SMT implementation can be updated to use the latest SAT technology as well as support many theories (any theory for which the satisfiability of conjunctions of ground theory literals is decidable), including such theories as Linear Integer Arithmetic, the theory of Arrays, Lists and other recursive data structures, theory of fixed width bitvectors and many others.

However, many verification tasks contain operators, terms and literals in several of the above theories simultaneously, possibly with non-disjoint signatures. A method that is commonly used to circumvent this problem is the Nelson-Oppen procedure for combining theories [NO79]. It is restricted to combinations of theories which have disjoint signatures (mixed signature literals are allowed), and for which any formula with a finite model has an model with an infinite universe. Many refinements to the original restriction have been proposed, e. g. , Jovanović and Barrett [JB11], most of which weaken the latter criterion.

Given that most decision procedures are over ground fragments of their respective theories, SMT solvers require extension to deal with quantified formulas. Methods proposed for quantifier reasoning fall into two categories: instantiation or finite model finding.

The key problem when instantiating quantifiers is finding relevant instances to check. E-matching, described in Detlefs et al. [DNS03], addresses this problem by finding subterms in the input formula which match with the context of the quantifier to be instantiated. For example, if $\forall x.\ L[t[x]] \wedge \phi$ is to be instantiated and there are terms $s \approx_E t[r]$ in ϕ, then $[x \to r]$ is used to produce a new instance. The process can be tuned by using a larger or smaller subset of the available contexts, called 'triggers', or these can be given by the user.

Ge and de Moura [GdM09] give a refinement to instantiation which guarantees completeness for certain universally quantified formulas. Specifically all quantified variables must occur as direct subterms of uninterpreted (i. e. non-theory) function

symbols. This approach is related to certain, theory specific, extensions of decidability to quantified fragments, such as the array properties fragment.

In the finite model finding camp, Reynolds et al. [RTGK13, RTG+13] give methods for finding finite models for formulas in which quantifiers range over uninterpreted types. This is distinct from the purely instantiation based method above, as it can find models for formulas which do not strictly fall into any completely instantiable fragment.

2.6.3 Princess

The Princess solver [Rüm08] operates in LIA extended with uninterpreted predicates. Specifically, it implements a free-variable sequent calculus with constraints; the constraints are discharged via the Omega test. Since only predicates are supported, all function symbols are relationally encoded with functionality and totality axioms added. All predicates are regarded as sets of integer tuples.

For this reason, the issue of sufficient completeness does not arise; the only first-order models under consideration are those which properly extend LIA. This does not immediately imply that Princess is complete in more cases, for example, Princess treats Skolemization differently than reasoners operating on clauses: it can introduce a new constant at any time for a nested existential, but may need to do this more than once for the same quantifier. Thus, some problems which Hierarchic Superposition would saturate but report 'unknown' due to a lack of sufficient completeness, Princess would not terminate on, as it attempts to define the whole range of the undefined terms.

In general, as for Hierarchic Superposition, Princess is complete on both pure first-order and pure arithmetic formulas. It is also complete for prenex normal form formulas with only universal or only existential quantifiers in the mixed signature.

Instead of abstracting formulas at the beginning of a proof, complementary instances of a predicate produce a LIA formula which must be checked, as follows:

$$\neg p(s_1, s_2, s_3) \wedge p(t_1, t_2, t_3) \text{ yields } s_1 \approx t_1 \wedge s_2 \approx t_2 \wedge s_3 \approx t_3$$

where s_i, t_i are pure $\Sigma_{\mathbb{Z}}$-terms and p is an uninterpreted predicate.

Princess also uses E-matching for quantifier instantiation, as used in SMT solvers. Although the calculus is complete without it, the use of user-defined trigger terms is an advantage [Rüm12].

Princess has performed well in CASC competitions [Sut15, Sut14], and also is used as a back-end for model checking applications.

2.6.4 SPASS+T

Waldmann and Prevosto [WP06] describe an extension of the SPASS solver (which implements saturation based reasoning using a Superposition calculus) that uses an SMT solver to implement theory reasoning. The two solvers are not as tightly integrated as in the case of Hierarchic Superposition: the first-order solver simply

passes to the SMT solver any clauses that it can handle (typically ground clauses with free function symbols and arithmetic symbols). The proof search terminates if either solver deduces a contradiction from its respective clause set. For some fragments (see note), all base formulas are guaranteed to be ground, and SPASS+T is complete in that case. For other fragments, the paper introduces an instantiation rule meant to generate the necessary ground clauses for the SMT solver. In addition, specialized arithmetic simplification rules are integrated in SPASS, either by input as axioms or hard coded rules.

Fore example, the integer ordering expansion rule:

$$\text{IEO}\ \frac{C \vee s \leq t}{C \vee s \approx t \vee s \leq t-1 \qquad C \vee s \approx t \vee s+1 \leq t}$$

is related to a rule used for recovering sufficient completeness described in Chapter 3. Prevosto and Waldmann note that it is obviously very productive but can be restricted to only apply to clauses with a single positive literal.

Although capable of proving unsatisfiability of problems (especially ground problems) over mixed theories, the combination is incomplete, and so a saturation does not guarantee the existence of a counter-example.

2.6.5 Nitpick

Nitpick [BN10] is a counter-example finder for higher-order logic (HOL), mainly used together with the Isabelle/HOL proof assistant. It translates HOL formulas to first-order relational logic (FORL), an extension of FOL with relational calculus operators, such as product, union and transitive closure. This language is implemented by Kodkod[TJ07], which in turn translates the problem to SAT.

In order to translate from HOL to FORL types in the HOL formula are restricted to finite cardinalities, called scopes. This applies also for arithmetic theories encoded in HOL which are treated specially by Nitpick. Specifically, only finite prefixes of $\mathbb{N}$ are used, the successor function is interpreted as a partial function (relationally encoded) as are functions which range over naturals. Partiality is soundly approximated in the translated formula, however, quantifiers ranging over naturals might not be disprovable. There is a mode in which such quantifiers are bounded, producing only hypothetical counter-examples.

Nevertheless, Nitpick is effective as a counter-example generator, and forms an important part of Isabelle's automation tool suite along with Sledgehammer and Quickcheck, which have complementary roles.

Chapter 3

Beagle – A Hierarchic Superposition Theorem Prover

3.1 Motivation

This chapter describes *Beagle*, an automated theorem prover for first-order logic modulo built-in theories. *Beagle* implements the Hierarchic Superposition calculus as described in Bachmair et al. [BGW94], Baumgartner and Waldmann [BW13b], and Chapter 2. Theory reasoning support is implemented for linear integer and linear rational arithmetic. *Beagle* features new simplification rules for theory reasoning and well-known ones used for non-theory reasoning. It also implements calculus improvements like *weak abstraction* [BW13b] and a method for determining (un)satisfiability w. r. t. quantification over finite integer domains. (Originally described in Baumgartner et al. [BBW14], this addition will be described in Chapters 5 and 6). *Beagle* is a test-bed implementation for those ideas.

Beagle is written in Scala and includes an implementation of a background reasoner for deciding fully quantified LIA formulas. Existing SMT solvers can be employed as background reasoners as well, via a textual SMT-LIB interface. *Beagle* accepts problem specifications written in the Typed First-order Formula (TFF) format (the typed version of the Thousands of Problems for Theorem Provers (TPTP) problem specification language) and in the SMT-LIB format [BST10].

This chapter describes the above features in more detail and reports on *Beagle*'s performance on benchmarks from the TPTP problem library [Sut09] and SMT-LIB[1]. It updates the previous system description [BBW15] with new results and descriptions of some new features.

Section 3.2 describes *Beagle*'s background reasoning components in general terms, giving an overview of how they relate to the first-order logic reasoning component. It also describes a generic minimization procedure used for dependency-directed backtracking when an unsatisfiable set of BG clauses has been found. Section 3.3 provides a detailed description of the *Beagle*'s LIA reasoner. Since most arithmetic problems found in the software verification domain use only LIA, this reasoner was chosen as the target of most of the optimization work. Performance results are given

[1] http://smtlib.cs.uiowa.edu/benchmarks.shtml

on a range of parametric problems in the pure LIA theory. The section also describes a novel solution extraction technique for Cooper's algorithm, that has the ability to return representative values for existentially quantified variables in satisfiable formulas. Section 3.4 describes the overall proof procedure insofar as it applies to the first-order parts of clauses. Finally, Section 3.5 describes *Beagle*'s performance on the TPTP and SMT-lib benchmark libraries, as well as reporting results from the yearly CASC competitions.

In this chapter 'BG clause' refers to a clause over one of the background theories $T_{\mathbb{Z}}, T_{\mathbb{Q}}$ or $T_{\mathbb{R}}$, and similarly for 'BG formula'. 'BG prover' refers to any of the built-in decision procedures for BG clauses, while 'proof procedure' refers to the Superposition based calculus.

A BG variable is either abstraction or general, it will be described as such if the distinction is important. Capital letters $\{X, Y, Z\}$ denote abstraction variables, and lower case letters $\{x, y, z\}$ denote ordinary variables.

There is a trade off between abstraction and ordinary variables: while ordinary variables enable 'more complete' theorem proving, they often lead to a larger search space. For example, the clause set $\{p(x), \neg p(c)\}$, where c is in the FG signature, is ($\mathcal{B}$-)unsatisfiable by virtue of the instance $p(c)$ of the first clause, and the prover will detect this. However, $\{p(X), \neg p(c)\}$ is $\mathcal{B}$-satisfiable, as the abstraction variable X is never instantiated with the FG constant c when forming the equivalent set of ground instances.

Although the usage of abstraction and general variables *within* a derivation is fixed, the implementation can choose either kind for BG variables in *input* formulas. Some proofs can only be found using general variables in input formulas, typically these problems do not have sufficient completeness. On the other hand, many proofs can be found using only abstraction variables in the input, and this strategy is much more efficient overall. *Beagle* supports both configurations, and switching between the two is a key step in the 'auto mode' described later.

3.2 Background Reasoning

Background reasoning is represented in *Beagle* as theory specific modules, *'solvers'*, that implement a specific interface (Fig. 3.2). This section describes the capabilities and uses of *Beagle*'s solvers.

At minimum, a theory solver must implement the Close inference rule given above, that is, it must decide the $\mathcal{B}$-satisfiability of sets of BG clauses. Hence, the solver must at least decide $\mathcal{B}$-satisfiability for the EA-fragment. If the BG clauses do not have free (BG-sorted) constants, they can be checked by a theory solver for the quantifier-free fragment. This case is rare however, so it is preferable to be able to decide $\mathcal{B}$-satisfiability in the EA-fragment to fully support quantified reasoning.

If the background theory admits quantifier elimination (QE), then problems in the EA-fragment can always be reduced to the universally quantified fragment and

```
// Implemented by individual solvers
def QE(cl: Clause): List[Clause]
def check(cls: Iterable[Clause]): SolverResult
// Compute a set of ground clauses that is equivalent
// to cl over the background domain.
def asSolverClauses(cl: Clause): (List[Clause], List[Clause])

// Implemented globally
def minUnsatCore(cls: Iterable[Clause]): List[Int]
def simplify()
def subsumes(cl1: Clause, cl2: Clause): Boolean
```

Figure 3.1: The Solver interface

checked with an efficient decision procedure, or can be discharged with a second round of quantifier elimination.

The Close check is used within *Beagle*'s proof procedure whenever a new BG clause is retained. This is an incremental process: the new clause is added to a set of BG clauses that is guaranteed to be $\mathcal{B}$-satisfiable. QE algorithms, such as Cooper's algorithm and Fourier-Motzkin, are not known to support incremental reasoning, though many decision procedures for the quantifier-free fragment do. Instead, a version of Cooper's algorithm is described in Section 3.3.2, which stores the bindings used in the outermost QE step. When applied to a valid EA-formula, this returns an assignment to the BG sorted constants in a model of the BG clauses. Re-applying this assignment before the next call to Close can produce a simpler, often trivially true, formula.

3.2.1 General Components

This section describes BG reasoning components common to all BG solvers used by *Beagle*. Examples will be assumed to be in extensions of $T_{\mathbb{Z}}$.

Quantifier elimination. Quantifier Elimination (QE) can be used for eliminating variables that only occur in BG literals of a non-BG clause. For example, the clause $p(x) \vee \neg(x < y) \vee \neg(y < 3)$ becomes $p(x) \vee \neg(x < 2)$ by QE of y from the subformula $\forall y.\ \neg(x < y) \vee \neg(y < 3)$. The general form of this transformation is

$$\text{QE-general}\ \frac{\forall \overline{x}.\ C[\overline{x}] \vee \forall \overline{y}.\ D[\overline{x}, \overline{y}]}{\forall \overline{x}.\ C[\overline{x}] \vee D'[\overline{x}]}$$

where D is a disjunction of BG literals, and D' follows from QE of the tuple of BG variables $\overline{y}$ from D.

However, using QE like this for clause simplification may destroy refutational completeness, since in general the result can be larger (under the clause ordering) than the clause being simplified. A special case is where the conclusion is $C \vee \top$, as

then the clause $C \vee D$ can safely be removed[2]. Checking all clauses with BG literals can be expensive, and is not necessary when all clause literals are pure BG.

Overall, this improvement does not make a large difference; the slight improvements in performance are balanced by losses elsewhere. In isolated cases where large clauses with trivial BG parts are deleted it can make a drastic improvement, but this also depends on other parameters being set correctly. For example, `SWW598=2` shows a 20s improvement, while `SWW619=2` shows a 25s loss in performance with this form of checking. Hence this optimization is disabled by default. Otherwise, *Beagle* uses this simplification only during preprocessing, which does not affect refutation completeness.

Splitting. *Beagle* optionally splits BG clauses into variable disjoint subclauses. If QE is available, then a version of each BG clause with the shared quantifiers instantiated is added to the current clause set, which is split exhaustively into unit clauses by *Beagle*'s splitting rule. For example,

Example 3.2.1. Take the clauses below, where $\mathcal{N}$ does not contain any further BG clauses.

$$0 \geq -3x + y \vee 0 \approx x - 5 \vee 0 > z, \mathcal{N}$$

The variable x shared between literals of the first clause can be eliminated using Cooper's algorithm. First, the clause is negated and literals with x are normalised:

$$\neg(\exists z, y, x.\ (3 \mid x \wedge 0 < -x + y \wedge 0 \not\approx x - 15 \wedge 0 \leq z))$$

The equivalent elimination formula:

$$\neg(\exists z, y.\ \bigvee_{j=1}^{3} (3 \mid (15 + j) \wedge 0 < -(15 + j) + y \wedge 0 \not\approx (15 + j) - 15 \wedge 0 \leq z))$$

Removing the outer negation:

$$\forall y, z.\ (0 < -16 + y \vee 0 > z) \wedge (0 < -17 + y \vee 0 > z)$$

Then splitting and simplification yields three clause sets:

$$0 < -16 + y, \mathcal{N}$$
$$0 < -17 + y, \mathcal{N}$$
$$0 < z, \mathcal{N}$$

Note that each has only unit BG clauses.

This is used only when the BG decision procedure only accepts ground unit clauses (equivalently, conjunctions of ground theory literals) as input.

[2]This is like the tautology deletion rule for SUP(LA) described in Chapter 2.

Simplification. Like with inference rules, simplification rules are prevented from applying to BG terms, e. g. , the unit clause $a + 5 \approx b + 2$ would not be used to rewrite inside another clause. BG simplification is useful in other cases, as, for example $a + 5 \approx a + 2$ can be shown to be unsatisfiable using cancellation rather than QE.

Beagle employs demodulation by BG tautologies and other forms of syntactic simplification for BG terms during a proof. These techniques must satisfy the usual conditions for simplification in the Hierarchic Superposition calculus to ensure completeness. However, as is often the case, *incomplete* strategies yield large performance gains.

In *Beagle* , simplification rules and strategies are classified as *cautious* and *aggressive*, the distinction being that cautious strategies preserve both sufficient completeness and refutation completeness, while aggressive ones may not. Cautious rules typically evaluate ground terms or eliminate obvious tautologies. Aggressive rules may eliminate double (arithmetic) negation or do algebraic cancellation, both of which may prevent future (necessary) inferences. The actual level of simplification (cautious or aggressive) can be set by the user or controlled internally.

Beagle also has a hard-coded set of theory specific simplification rules which act on arithmetic terms and literals. Unlike lemmas, BG simplification rules do not appear as part of the clause set. The following subsections describe theory specific simplification rules.

Beagle removes disequations of certain forms from clauses by *unabstraction*. This is effectively the inverse of an abstraction step, although the abstracted term may have been modified since it was abstracted, e. g. , by demodulation to a domain element. The general form of the unabstraction rule is:

$$\text{Unabstract}\ \frac{C \vee x \not\approx t}{C[x \to t]}$$

where x is BG-sorted and does not occur in t.

This is similar to the usual equality resolution rule from the Superposition calculus, however it applies only to BG sorted literals of a specific form, therefore is classified as a form of BG reasoning.

Unabstraction has cautious and aggressive variants: for example, if cautious simplification is chosen, literals of the form $x \not\approx d$ are removed by unabstraction only if d is a concrete number.

Aggressive unabstraction allows t to be *any* term, including FG terms. It can break completeness, since there is no guarantee that the unabstracted clause is smaller than all possible simple ground instances of the abstracted clause.

Example 3.2.2. Let $C = f(x) \approx 0 \vee x \not\approx a + 5$ where a is a parameter, then C produces $f(a + 5) \approx 0$ by unabstraction. The clause $f(0) \approx 0 \vee 0 \not\approx a + 5$ is in $\text{sgi}(C)$ and since $0 \prec a + 5$ in the term ordering it follows that $f(0) \approx 0 \vee 0 \not\approx a + 5 \prec f(a + 5) \approx 0$. So the result of unabstraction does not make the original clause redundant in this case .

As for other simplification rules, the specific level of unabstraction is controlled internally and, typically, only the results of *cautious* unabstraction are kept. Aggressive unabstraction is used to derive unit clauses which may demodulate other clauses, but only those demodulation results are used, not the unit clauses resulting from unabstraction. In general, clauses (unit or otherwise) produced by unabstraction are never directly added to the set of retained clauses, but can be used in satisfiability checks.

Lemmas. For common, but undecidable, extensions to BG theories (non-linear arithmetic in particular), *Beagle* uses lemmas and built-in operators to provide best-effort support. Non-linear arithmetic terms $x \cdot y$ are translated by replacing the product operator with a new uninterpreted (FG) operator $\mathsf{prod}_{\mathbb{Z}}$. Axioms are included to define $\mathsf{prod}_{\mathbb{Z}}$ in terms of BG operators $0, 1, +_{\mathbb{Z}}$, while simplification rules replace any linear $\mathsf{prod}_{\mathbb{Z}}$-terms with their BG equivalent. Extra *lemmas* about $\mathsf{prod}_{\mathbb{Z}}$ are also included, e. g. , the usual distribution and commutation laws, most of which can only be proven by induction on the definitions. All of the $\mathsf{prod}_{\mathbb{Z}}$ lemma formulas used are listed below:

(1) $\mathsf{prod}_{\mathbb{Z}}(0, x) \approx 0$
(2) $\mathsf{prod}_{\mathbb{Z}}((1 + x), y) \approx y + \mathsf{prod}_{\mathbb{Z}}(x, y)$
(3) $-\mathsf{prod}_{\mathbb{Z}}(x, y) \approx \mathsf{prod}_{\mathbb{Z}}(-x, y)$
(4) $\mathsf{prod}_{\mathbb{Z}}(x, y) \approx x \Leftrightarrow (x \approx 0 \vee y \approx 1)$
(5) $(0 < x \wedge 0 < y) \Rightarrow 0 < \mathsf{prod}_{\mathbb{Z}}(x, y)$
(6) $\mathsf{prod}_{\mathbb{Z}}(x, (y + z)) \approx \mathsf{prod}_{\mathbb{Z}}(x, y) + \mathsf{prod}_{\mathbb{Z}}(x, z)$
(7) $\mathsf{prod}_{\mathbb{Z}}(x, y) \approx \mathsf{prod}_{\mathbb{Z}}(y, x)$

Formulas (1) and (2) define $\mathsf{prod}_{\mathbb{Z}}$, (6) and (7) give basic algebraic properties only provable via induction and (3)-(5) are useful algebraic simplifications also difficult to prove otherwise. The associative law was left out due to severe performance degradation on many examples.

'Lemma' in this context simply denotes a valid formula which is treated specially by the proof procedure; as for the *set-of-support* strategy in resolution solvers, inferences between lemma clauses are disallowed. Unlike set-of-support, there is no expectation that the set of lemmas is saturated, they are simply extra clauses which might help in a derivation, but might otherwise be needlessly over-productive (e. g. , associativity and commutativity). An advantage of this arrangement is that lemmas subsume identical clauses in the input, providing the user with a way to prune over-productive clauses.

Solvers. *Beagle* implements solvers for linear integer arithmetic (LIA) and linear rational arithmetic (LRA). It also accepts linear real arithmetic, but the differences are merely syntactic. Alternatively, existing SMT solvers can be coupled via a textual SMT-LIB interface. In addition, *Beagle* can make use of minimal unsatisfiable cores, that can be produced by SMT solvers such as Z3 [dMB08]. Unsatisfiable cores can be exploited for dependency-directed backtracking, described in the next section.

3.2.2 Minimal Unsatisfiable Cores

When the Close rule applies to a set of BG clauses D, *Beagle* determines a minimal unsatisfiable subset of D (a minimal unsatisfiable core (MUC)). This core is used for *dependency-directed* backtracking of split levels.

A split level is a set of clauses whose derivation includes the left conclusion of a split inference. Each split conclusion corresponds to a branch in the proof search space.

Definition 3.2.1 (Split Level)**.** Split level S_0 is the set of input clauses. Split level S_{n+1} consists of the left conclusion of a split inference on a clause in split level S_n and any conclusion of an inference whose premises contain a clause in S_{n+1}.

When $S_n \models \square$ is concluded, the proof procedure can backtrack to S_m where $m < n$, by removing any clauses in split levels higher than m from the current clause set and then adding the right conclusion of the last split.

The unsatisfiable subset is minimal w. r. t. unsatisfiability, i. e. , any proper subset is satisfiable. Then, by backtracking to the level before the split that produced the *maximal* split level in the unsatisfiable set, the new split level will not contain the same BG clauses that caused unsatisfiability of the clause set before backtracking.

Currently, minimal unsatisfiable subsets are found by applying a simple minimization algorithm to the unsatisfiable BG clause set, or by using the built-in unsatisfiable core algorithm in the Z3 SMT-solver [dMB08].

An unsatisfiable clause set can have multiple minimal unsatisfiable cores. For example, let unit clauses P, Q be in S_0, $\neg P \in S_1$ and $\neg Q \in S_2$. The clause set $S_0 \cup S_1 \cup S_2$ has two minimal unsatisfiable cores: $\{P, \neg P\}$ and $\{Q, \neg Q\}$. Depending on which is selected, either S_1 or S_0 might be backtracked to. Of course, backtracking to S_1 is not helpful as this still derives the unsatisfiable set $\{P, \neg P\}$. Therefore, the maximal split level of clauses in the unsatisfiable core should also be minimized, otherwise the heuristic will not be effective.

Lemma 3.2.1. *Given an unsatisfiable set of BG clauses $\mathcal{N}$, the algorithm* minUnsatCore *finds a MUC $T \subseteq \mathcal{N}$ such that the maximal split level of any clause in T is less or equal to the maximal split level in any other MUC of $\mathcal{N}$.*

Proof. Assume the contrary, that some MUC $S \subseteq \mathcal{N}$ is returned whose split level (i. e. , the largest split level of any clause in M) is larger than the split level of some other MUC $T \subseteq \mathcal{N}$. By assumption, both S and T are non-empty, and so the maximal $c \in S$ w. r. t. split level has higher split level than any clause in T. It is an invariant that the list `cs` remains sorted after all removals. Therefore, c has a lower index in `cs` than every clause in T, and so a clause set including T but excluding c is checked before any clause of T can be removed. Since this clause set contains the MUC T, it is unsatisfiable, and so the clause c is removed from `cs`. Then S is not returned by minUnsatCore. □

Testing on TPTP-v6.1.0 shows that this heuristic provides generally good results. The best performance improvement was 44s, while the worst degradation was only 2.8s.

```
1  algorithm minUnsatCore(cls: List[Clause]):
2    if (check(cls) is SAT):
3      Nil
4    else:
5      let i = 0
6      let cs = sortDescSplitLevel(cls) //the working clause set
7      while (i < cs.size):
8        if (check(cs.drop(i)) is UNSAT):
9          cs = cs.drop(i)
10         i = 0
11       else i += 1
12
13     return cs //no more removals possible
```

Figure 3.2: Pseudocode for MUC algorithm

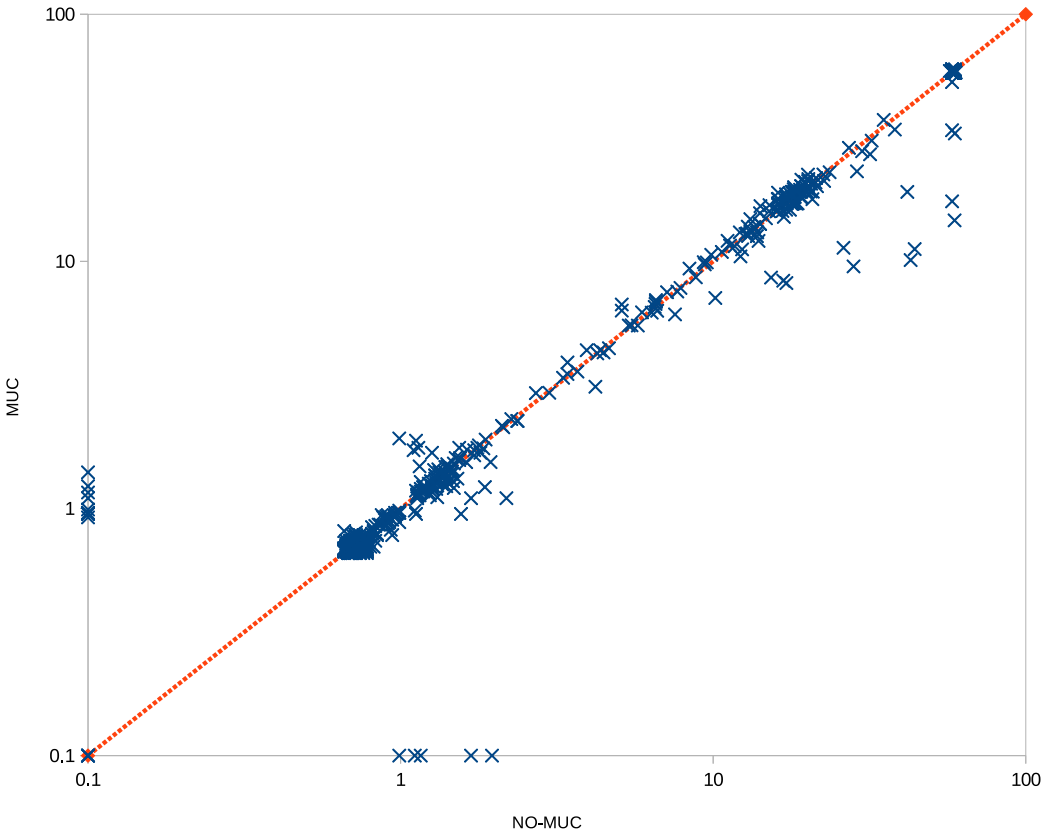

Figure 3.3: Run time in seconds of *Beagle* with and without MUC

Figure 3.2.2 shows the performance of *Beagle* (wall-clock time in seconds) on all Typed First-order Arithmetic (TFA) problems in the TPTP-v6.1.0 library. The vertical axis measures run time with the MUC optimization enabled and the horizontal shows run time without. Both axes are logarithmic.

Of the 954 LIA problems tested, 837 problems showed no performance change either way (these problems are typically solved without backtracking), and 117 problems showed a performance difference (a shift of more than 0.5s). Roughly half (59) performed better, and 58 performed worse with MUC enabled. However, many of the problems which showed improved performance with MUC enabled outperformed the regular version by at least ten seconds, while the worst performers lost no more than 3 seconds. The best improvement was 45s on NUM860=1, while NUM862=1 and SWW650=2 showed significant improvement. As a result, MUC is enabled by default.

3.2.3 Other Arithmetic Features

Linear Rational Arithmetics. The solver for LRA comprises a Fourier-Motzkin[3] style QE procedure for eliminating BG variables. This eliminates variables from DNF $T_{\mathbb{Q}}$-formulas by replacing $s_i \leq x$ with $s_i \leq t_j$ for each literal $x \leq t_j$ appearing in the same disjunct as $s_i \leq x$. The variable x is eliminated from a particular disjunct after all such pairs are added. However, this leads to a worst case double exponential growth in the size of formulas [Mon10], so once the formula has been reduced to a single quantifier alternation (i. e., of the form $\overline{\exists}.\ F$) a Simplex solver is used to eliminate the final varaibles.

This solver is an off-the-shelf implementation of the Simplex algorithm[4]. In order to support literals with strict inequalities, an extra variable is introduced. For example, $ax + by + cz > k$ for $a, b, c, k \in \mathbb{Q}$ and variables x, y, z, becomes $ax + by + cz \geq k + d$ assuming $d > 0$, where d is a new variable. The new variable d is reused for all inequalities, and to satisfy the constraint $d > 0$ the value of d is maximized by the Simplex algorithm. If a solution exists but $d \leq 0$ after maximizing, then the problem is unsolvable ($T_{\mathbb{Q}}$-invalid), otherwise it is $T_{\mathbb{Q}}$-valid.

The cautious simplification rules for LRA evaluate arithmetic subterms, and the aggressive simplification rules rewrite sub-terms towards a flat structure by exploiting AC-properties of the operators as for LIA. Syntactic differences between concrete numbers aside, linear real arithmetic is treated by additional lemmas that are valid in real arithmetic. Overall, the LRA solver is not as advanced as the LIA solver.

Non-linear Arithmetic. *Beagle* features a simplistic treatment of non-linear arithmetic. During preprocessing, every occurrence of a non-linear multiplication subterm $s \cdot t$ is replaced by $\mathsf{prod}(s, t)$, where prod is a dedicated foreground function

[3] Due to J. Fourier, 1824. A description of the method's application as a decision procedure is in Monniaux [Mon10]

[4] Part of the Apache Commons math library. See http://commons.apache.org/proper/commons-math/

symbol of the proper arity. As soon as s or t in $\mathsf{prod}(s, t)$ is replaced by a concrete number, the resulting term is turned into a (theory) multiplication term again. There are dedicated lemmas for each of the theories $T_{\mathbb{Z}}, T_{\mathbb{Q}}, T_{\mathbb{R}}$, that define multiplication in terms of repeated addition and specify other difficult to prove properties of multiplication. See the previous section on lemmas in Section 3.2.1. An alternative, described in Chapter 4, is to attempt to prove that the input formula ϕ is $\mathcal{B}$-unsatisfiable (as opposed to proving $\neg\phi$ unsatisfiable). If ϕ contains no uninterpreted terms other than non-linear product terms, then the fact that $\neg\phi$ is $\mathcal{B}$-valid follows from the unsatisfiability of ϕ and the satisfiability of the axioms defining the product operator. This observation was useful in the competition, see Section 3.5.3, but requires some care to apply correctly.

Chaining. The optional chaining inference rules apply the transitivity property of $<$. One of them is *positive chaining*:

$$\text{Positive chaining}\ \frac{s < t \vee C \qquad u < v \vee D}{\mathsf{abstr}((s < v \vee C \vee D)\sigma)}$$

if σ is an mgu of t and u.

Other chaining rules deal with negative inequations $\neg(u < v)$ in the right premise. Currently, the only restriction is that the literals selected for inferencing are not pure BG.

A variation on the chaining rule can be used to recover sufficient completeness in certain cases. Consider a problem of the form:

Example 3.2.3. Let x, y be $\mathbb{Z}$-sorted variables, a some $\mathbb{Z}$-sorted constant, and $f : \mathbb{Z} \to \mathbb{Z}$.

$$(1)\quad a < f(x) \qquad\qquad (2)\quad f(x) < a + 4$$
$$(3)\quad (0 \leq x \wedge x \leq 3) \Rightarrow f(x) < f(x+1)$$

The set of (1), (2), and (3) is inconsistent, since (1) and (2) allow f to take at most three distinct values, while (3) requires four.

Unsatisfiability can be demonstrated in the Hierarchic Superposition calculus by introducing either of

$$(4.1)\ f(x) \approx a + 1 \vee f(x) \approx a + 2 \vee f(x) \approx a + 3$$
$$(4.2)\ \{f(0) < f(1), f(1) < f(2), f(2) < f(3)\}$$

Adding (4.1) to the clause set recovers sufficient completeness, as then any model must satisfy $f(t) \approx a + 1, f(t) \approx a + 2$, or $f(t) \approx a + 2$ where t is ground. On the other hand, adding (4.2) does not immediately give sufficient completeness, but it adds new instances of f to the clause set which may not otherwise occur. These instances enable the derivation of the required contradiction.

The reasoning in the above example can be formalized by an inference rule that

generalizes the LIA theorem

$$\exists a : \mathbb{Z}.\ (a < t \wedge t < a + k) \Leftrightarrow \bigvee_{j=1}^{k-1} t \approx a + j \tag{3.1}$$

where $k > 1 \in \mathbb{N}$, and t is any integer sorted term.

There are two forms of this rule, corresponding to the left and right directions of (3.1).

$$\text{Inst-Right}\ \frac{r < s \vee C \qquad t < u \vee D}{(s \approx r + 1 \vee \ldots \vee s \approx r + (k-1) \vee C \vee D)\sigma}$$

$$\text{Inst-Left}\ \frac{\neg(r < s) \vee \neg(t < u) \vee C}{(s \not\approx r + 1 \vee C)\sigma,\ \ldots,\ (s \not\approx r + (k-1) \vee C)\sigma}$$

where $\sigma = \text{mgu}(s, t)$; $(u - r)\sigma \approx_{\mathbb{Z}} k \in \mathbb{N}$, $k > 1$; both $r < s$ and $t < u$ are not pure-BG.

Sufficient completeness (of a subset of clause instances) is recovered in the special case where both C and D are empty in the premises of Inst-Right. As with the Define rule, Inst-Right is best applied eagerly in a derivation. The Inst-Left rule does not recover sufficient completeness, although it does introduce clause instances which could be useful in a derivation, as seen in the example.

This illustrates an interesting overlap of theory reasoning and sufficient completeness. Where a clause set has sufficient completeness already, Inst-Right and Inst-Left are not necessary, as all clauses with free BG sorted terms are eventually equivalent to some BG clauses. On the other hand, for clause sets without sufficient completeness rules that implement theory reasoning for free BG terms can allow derivations which would otherwise not be possible. Theory reasoning, as a strategy for dealing with a lack of completeness, has the advantage of being applicable to all clause sets extending that particular background theory.

3.3 Linear Integer Arithmetic

As previously mentioned, the solver for the LIA theory in *Beagle* is a custom implementation of Cooper's algorithm. Satisfiability in the EA-fragment of $\Sigma_{\mathbb{Z}}$ is decided by two rounds of QE.

A high level description of the essentials of Cooper's algorithm as implemented in *Beagle* , following Harrison [Har09], is given below:

Let $\exists x.\ F$ be a formula in negation normal form, where F is quantifier-free. The aim of any QE algorithm is to produce from F some quantifier-free formula G that is equisatisfiable w. r. t. the given theory, $T_{\mathbb{Z}}$ in this case. Note that, in general, universal quantifiers are presumed to be eliminated using the equivalence $\exists x.\ F \Leftrightarrow \neg\forall x.\ \neg F$.

The following assumes primitive operators $\approx, <, \not\approx, |$, so all literals of F must be translated to one of these, e. g. , $s \geq t$ becomes $t < s \vee t \approx s$. Every literal of F is assumed to be normalized into either of the forms

1. $0 \bowtie c_1 x_1 + \ldots + c_n x_n + k$, where $\bowtie \in \{\approx, <, \not\approx\}$ or,
2. $d \mid c_1 x_1 + \ldots + c_n x_n + k$, for $d \in \mathbb{Z}$ and possibly negated.

where the coefficients c_i and k are concrete integers whose greatest common divisor is 1

Definition 3.3.1 (Cooper's Algorithm)**.** To eliminate x from $\exists x.\, F[x]$, do the following:

1. Let l be the lcm of all x coefficients in F and replace literals as follows
   - replace $0 \approx cx + t$ with $0 \approx x + (l/c)t$
   - replace $0 < cx + t$ with $0 < x + |l/c|t$, or $0 < -x + |l/c|t$ if c is negative
   - replace $d \mid cx + t$ with $d \mid x + (l/c)t$

   Similar for negated versions of literals. Let $\mathsf{unit}_x(F) = F' \wedge l \mid x$, where F' is F with all literals transformed as above.
2. Let $F_{-\infty}[x]$ be the formula that results from replacing literals $0 \approx x + t$, $0 < x + t$ with $\bot$, and replacing literals $0 < -x + t$, $0 \not\approx x + t$ with $\top$ in $\mathsf{unit}_x(F)$.
3. Let B_x be the set such that
   (a) $-t \in B_x$ if either $0 < x + t$ or $0 \not\approx x + t$ occurs in $\mathsf{unit}_x(F)$, and
   (b) $-(t+1) \in B_x$ if $0 \approx x + t$ occurs in $\mathsf{unit}_x(F)$

   Let $F_B[j] := \bigvee_{b \in B_x} \mathsf{unit}_x(F)[b + j]$ for a fresh variable j.
4. Let D be the lcm of all of the literals $d \mid x$, or $\neg(d \mid x)$ in $\mathsf{unit}_x(F)$, or 1 otherwise. Cooper's theorem establishes that

$$\exists x.\, F[x] \Leftrightarrow \bigvee_{j=1}^{D} (F_{-\infty}[j] \vee F_B[j]) \tag{3.2}$$

The right-hand formula in the equivalence (3.2) is called the *elimination formula* for x.

Example 3.3.1. Consider the elimination of x from

$$F = \exists y, x.\, 0 < -3x + y \wedge 0 \not\approx y - 5$$

Note that F is already in normal form.

$$\begin{aligned}\mathsf{unit}_x(F) &= 3 \mid x \wedge 0 < -x + y \wedge 0 \not\approx y - 5\\ l_x &= 3\\ F_B &= \bot, \text{ since } B_x = \emptyset\\ F_{-\infty} &= (3 \mid j \wedge \top) \wedge 0 \not\approx y - 5\end{aligned}$$

Then

$$F \Leftrightarrow \exists y.\ \bigvee_{j=1}^{3} (3 \mid j \wedge 0 \not\approx y - 5)$$

Quantifier elimination. The built-in LIA solver is based on Cooper's algorithm as given above, and includes improvements as introduced in Cooper [Coo72]. It accepts arbitrary BG formulas, in particular conjunctions of clauses. The code roughly follows the algorithm described in Harrison [Har09]. The LIA solver is used for both deciding satisfiability of sets of BG clauses (Close rule) and for the elimination of variables as described above (QE-general rule).

The implementation includes several improvements to Cooper's algorithm to make it more practical:

- conjunctions such as $x < 5 \wedge x < 3$ are replaced by $x < 3$, a limited form of subsumption.
- variables that admit unbounded (above or below) solutions are eliminated, e. g. , $\exists x.\ x \not\approx 0 \wedge F$ where x does not occur in F, is equivalent to F.
- elimination of equations $x \approx t$ where x does not occur in t, is accomplished by substitution of t for x.

Furthermore, if a conjunction contains the atomic formulas $s_1 < \alpha, \ldots, s_m < \alpha$ and $\alpha < t_1, \ldots, \alpha < t_n$, given that α does not occur elsewhere, then α can be removed by exhaustive resolution. (Resolution of $s < \alpha$ and $\alpha < t$ yields $s + 1 < t$.) If α does occur somewhere else, then this form of resolution can still be used to prove unsatisfiability when $s + 1 < t$ is false. This is similar to the first step of the Omega test for deciding Presburger arithmetic [Pug91].

The improvements mentioned above often help to solve problems much faster.[5] However, most are effective only on conjunctions of literals. To maximize their utility, the implementation deviates from the standard Cooper algorithm by multiplying out disjunctions in the RHS of (3.2). This can avoid deeply structured 'or-and' formulas and, as a special case, disjunctive normal form is preserved by solving and multiplying out the conjunctions separately.

Specifically, input to the algorithm is assumed to be a disjunction $F_0 = \exists x.\ G_0 \vee G_1 \vee \ldots \vee G_k$ where each G_i is a conjunction. Each disjunct G_i is treated separately,

[5] E.g., the GEG-problems in the TPTP problem library.

this well-known *block-elimination* enhancement reduces the lcm of the x coefficients. The simple elimination tests are applied first (for all variables, not just x), then the elimination formula for G_i is produced. Assume $G_i = G_i^x \wedge G_i'$ such that G_i' does not contain x and that the result of (3.2) is $H_0 \vee H_1 \vee \ldots \vee H_l$, then $G_i = (H_0 \vee H_1 \vee \ldots \vee H_l) \wedge G_i'$. This is the formula that is multiplied out, and Cooper is called recursively on $(H_0 \wedge G_i') \vee \ldots \vee (H_l \wedge G_i')$.

The final step of Cooper's algorithm involves instantiation over representatives of congruence classes of solutions for the target variable, which quite often leads to prohibitively large formulas. Using an improvement suggested in Harrison [Har09], *Beagle* occasionally defers this instantiation (based on the expected number of instances) until a later round of quantifier elimination. This is done by substituting a fresh variable and terms that describe the solution range, as occasionally a shorter proof of satisfiability/unsatisfiability can be found using a different variable.

Simplification and arithmetic terms normalization. The cautious simplification rules for LIA comprise evaluation of arithmetic terms, e. g. , $3 \cdot 5$, $3 < 5$, $\alpha + 1 < \alpha + 1$ (equal LHS and RHS terms in inequations), and rules for TPTP-operators, e. g. , $\mathsf{\$to\_rat}(5)$, $\mathsf{\$is\_int}(3.5)$. For aggressive simplification, integer sorted subterms are brought into a polynomial-like form and are evaluated as much as possible. For example, the term $5 \cdot \alpha + \mathsf{f}(3 + 6, \alpha \cdot 4) - \alpha \cdot 3$ becomes $2 \cdot \alpha + \mathsf{f}(9, 4 \cdot \alpha)$. BG formulas always produce proper polynomials, which can be used directly by the QE procedure without further conversions.

Aggressive simplification does not always preserve sufficient completeness. For example, in the clause set $\mathcal{N} = \{p(1 + (2 + \mathsf{f}(x))),\ \neg p(1 + (x + \mathsf{f}(x)))\}$, the first clause is aggressively simplified, giving $\mathcal{N}' = \{p(3 + \mathsf{f}(x)),\ \neg p(1 + (x + \mathsf{f}(x)))\}$. Notice that both $\mathcal{N}$ and $\mathcal{N}'$ are $T_{\mathbb{Z}}$-unsatisfiable, $\mathrm{sgi}(\mathcal{N}) \cup \mathrm{GndTh}(T_{\mathbb{Z}})$ is unsatisfiable, but $\mathrm{sgi}(\mathcal{N}') \cup \mathrm{GndTh}(T_{\mathbb{Z}})$ is satisfiable, since $1 + (2 + f(2)) \approx (1 + 2) + f(2)$ is not a theorem of $\mathrm{GndTh}(T_{\mathbb{Z}})$. Thus, $\mathcal{N}$ is (trivially) sufficiently complete while $\mathcal{N}'$ is not.

Aggressive simplification also includes heuristics for normalizing equations and inequations. Inequations are normalized by first eliminating the operators $>$, $\geq$ and $\leq$ in terms of $<$. The QE procedure treats $<$ as a primitive, so this is a natural choice. Then, the monomials of the LHS and RHS polynomials are moved around so that only positive signs and only addition of monomials (not subtraction) results. The rationale is to normalize terms by removing unnecessary operators. Similar heuristics apply for equations, which attempt to produce orientable equations. For example, $f(x) + 1 \approx g(y) + 2$ is not orientable, but $f(x) - g(y) \approx 1$ is, as 1 is smaller that any FG term in the term order. Normalizing (in)equations may remove or install sufficient completeness and destroy refutational completeness. Yet, experiments showed that aggressive simplification is far superior to cautious simplification in practice, hence it is enabled by default.

3.3.1 Performance

Although there are many high-level descriptions of Cooper's algorithm implementations, there are few descriptions of actual implementation details, for example Phan and Hansen [PH15] describe an implementation optimized for parallelism. For testing their implementation, the authors used a parametric form of the pigeon hole problem encoded in Peano Arithmetic.

This section describes a selection of parametric problems in the language of $\Sigma_{\mathbb{Z}}$ and the performance of *Beagle*'s Cooper solver on them. There are five problem classes, one of which is encoded in two ways. Table 3.1 reports the results of *Beagle* and CVC4 (version 1.4) on the problem instances, along with the parameters used and the problem's satisfiability status. The following sections describe each problem class along with the meaning of their parameters. In general, the problems instances reported in Table 3.1 were chosen to show points where the performance of either solver changed, or to illustrate an apparent relationship between some parameter and the solving time.

These experiments were carried out on a Linux desktop with a quad-core Intel i7 chip running at 2.8 GHz, with 8GB of RAM, although the host JVM[6] was configured with maximum heap size of 4GB (relevant for *Beagle*). The CPU time limit was 60 seconds soft (solver's heuristic target time) and 65 seconds hard (unresponsive processes killed).

The values in the status column reflect the *expected* result of a proof attempt, based on the construction of the problem. They have the following meanings:

- "Theorem/Counter-Sat" results indicate that the formulas have a designated conjecture formula which will be negated by the solver.
- "Satisfiable/Unsatisfiable", have no designated conjecture.
- "?" indicates that status of the problem is unknown.

The rationale behind comparing with CVC4 is that CVC4 implements projection style BG reasoning []. As can be easily observed from the table, CVC4's implementation is far more sophisticated than that of *Beagle*, however, the class of problems for which QE is suited is not completely subsumed by projection style reasoning.

Systems of Linear Equations. Given equations

$$\begin{aligned} a_{00}x + a_{01}y + a_{02}z &\approx 0 \\ a_{10}x + a_{11}y + a_{12}z &\approx 0 \\ a_{20}x + a_{21}y + a_{22}z &\approx 0 \end{aligned}$$

for fixed integer coefficients a_{ij}, check if there exists an assignment to the variables that satisfies all equations. There can be either no solution, a single solution or infinitely many solutions, depending on the choice of coefficients. Cooper's algorithm

[6] OpenJDK v.1.8

Problem	Parameter	Status	Cooper	CVC4
Frobenius	$S = \{7, 8\}$	Satisfiable	1.31	-
	$S = \{17, 18\}$	Satisfiable	1.11	-
	$S = \{34, 35\}$	Satisfiable	3.14	-
	$S = \{11, 17, 25\}$	Satisfiable	5.60	-
	$S = \{53, 24, 27\}$	Counter-Sat	4.74	-
	$S = \{179, 89, 90\}$	Satisfiable	-	-
	$S = \{3, 11, 17, 25\}$	Satisfiable	-	-
nQueens	n = 3	Unsatisfiable	0.1	0
	n = 4	Satisfiable	1.29	0.01
	n = 8	Satisfiable	31.91	0.03
Subset-sum	$\mid S \mid = 2, n = 111, k = 5$ (1)	Theorem	0.1	0.01
	$\mid S \mid = 2, n = 111, k = 5$ (3)	Counter-Sat	1.1	0.01
	$\mid S \mid = 3, n = 13, k = 3$ (1)	Counter-Sat	0.93	0
	$\mid S \mid = 5, n = 55, k = 7$ (1)	?	-	-
Pigeon-Hole Ex.	p = 5, h=6	Satisfiable	0.88	0
	p = 7, h=6	Unsatisfiable	10.15	0.62
	p = 10, h=9	Unsatisfiable	-	-
	p = 10, h=11	Satisfiable	1.78	0.01
Pigeon-Hole Rel.	p = 5, h=6	Satisfiable	2.87	0
	p = 7, h=6	Unsatisfiable	12.96	0.1
	p = 10, h=9	Unsatisfiable	-	1.33
	p = 10, h=11	Satisfiable	8.17	0.8
Linear Equations	n=3	Satisfiable	0.78	0
	n=3	Satisfiable	11.98	0
	n=3	Satisfiable	-	0
	n=3	Unsatisfiable	0.79	0
	n=3	Unsatisfiable	25.1	0
	n=5	Unsatisfiable	-	0

Table 3.1: Cooper performance on representative instances of problems

is especially sensitive to the size of coefficients, hence choosing larger coefficients provide good test cases for the instantiation phase.

Run time is proportional to the lcms of the coefficients, and it doesn't appear to matter whether it is satisfiable or unsatisfiable. The exception is where one equation is a constant multiple of another, this case can be easily detected.

CVC4 has a built-in linear Diophantine equation solver, which likely explains the excellent performance on this problem set.

Frobenius problem. Given a set of k positive numbers $\{a_1, \ldots, a_k\}$ whose gcd is 1, find the maximum number that cannot be expressed as a sum $a_1x_1 + \ldots + a_kx_k$ for positive x_i. For set $\{11, 17, 25\}$, the problem is equivalent to showing the following

formula is satisfiable:

$$\begin{aligned}\exists y.\ (\forall k_1, k_2, k_3.\ & \\ & ((0 \leq k_1 \wedge 0 \leq k_2 \wedge 0 \leq k_3) \Rightarrow \\ & y \not\approx 11 \cdot k_1 + 17 \cdot k_2 + 25 \cdot k_3)) \wedge \\ \forall x.\ (\forall k_1, k_2, k_3.\ & \\ & ((0 \leq k_1 \wedge 0 \leq k_2 \wedge 0 \leq k_3) \Rightarrow \\ & x \not\approx 11 \cdot k_1 + 17 \cdot k_2 + 25 \cdot k_3)) \Rightarrow x \leq y))).\end{aligned}$$

The difficulty of the problem can be adjusted by changing k or the values a_i in the set $\{a_1, \ldots, a_k\}$.

Problems in the table simply check the formula above with the set of coefficients given. The final problem is counter-satisfiable (i. e. , the negation of the above formula is satisfiable), as the parameters do not have gcd 1.

There are analytic solutions for $k = 2$ and $k = 3$. The performance of Cooper grows with the Frobenius number, which at least for $k = 2$ and $k = 3$ is proportional to $a_1 \times \ldots \times a_k$. However, the instance $\{3, 11, 17, 25\}$ has Frobenius number 19 yet it is not solved, suggesting that the difficulty also scales with k.

Subset sum game. Consider a two player game, where given a set of non-zero numbers S and number n, each player alternately subtracts a value in S from n until 0 is reached. Values in S are not removed during play. A player wins when they reach exactly 0, and loses if forced to subtract a value making the running sum negative. Hence, a player can also win if they force the other player to make a losing move. The problem is to show that given a fixed set S and positive numbers n, k there is a winning strategy for the first player in k steps. Expressed as a first order formula, for $S = \{1, 3, 4\}$ and $n = 11, k = 3$:

$$\begin{aligned}\exists x_1.\ (((x_1 \approx 1 \vee x_1 \approx 3 \vee x_1 \approx 4) \wedge 11 - x_1 \geq 0) \wedge & \\ \forall y_1.\ ((y_1 \approx 1 \vee y_1 \approx 3 \vee y_1 \approx 4) \wedge (11 - x_1 - y_1) \geq 0)) \Rightarrow & \\ \exists x_2.\ ((x_2 \approx 1 \vee x_2 \approx 3 \vee x_2 \approx 4) \wedge (11 - x_1 - y_1 - x_2) \approx 0) &\end{aligned}$$

Although there are other, more effective algorithms for proving the existence of winning strategies, the value in this problem lies in the fact that the number of quantifiers can be adjusted by setting the number of steps k.

Problem instances listed above have parameters $|S|$, n, k, where values in S are chosen from the range $[1, \lfloor n/2 \rfloor]$ and k must be odd. Instances are allowed to be infeasible, e. g. , if $k \cdot \mathsf{max}(S) < n$.

Problem difficulty appears to scale with the number of possible move sequences, roughly $|S|^k$. For example, where both $|S|$ and k are small, the problems are easily solved regardless of the size of n. (In fact if n is too large then the problem is always infeasible). Conversely, if both $|S|$ and k are large, then problems become difficult, even if n is small.

SAT problems. Boolean SAT problems can be encoded, simply by replacing each Boolean variable x with the literal $x' \geq 0$, or by $x' \approx 1$ and then adding $x \approx 0 \vee x \approx 1$. Checking satisfiability of the SAT problem is equivalent to checking satisfiability of the existential closure of the $\Sigma_{\mathbb{Z}}$-formula. Certain common SAT problems have more efficient encodings. The test problems include two encodings of the pigeon-hole problem. The first uses integer-sorted variables one for each 'pigeon', and restricts the values each variable may take to be in $[0, h-1]$ (h is the number of holes). The variable takes the value of the hole the pigeon is in. This is the existential (Ex.) encoding in the table. The second encoding uses a simple Boolean encoding where each Boolean variable ($p_{x,y}$ means pigeon x is in hole y) is replaced with $x \geq 0$. In the table, parameter p is the number of pigeons and h the number of holes.

There is also an encoding of the n-queens problem with integer-sorted variables, where the ith variable represents the column position of the queen in the ith row.

All such SAT problems are limited to a single quantifier. The only adjustable parameter is the number of variables and possible assignments.

The results show better performance of the Cooper solver on the existential encoding with faster run times for each given parameter. As typical for pigeon-hole problems, satisfiable instances are solved much more easily than unsatisfiable ones for similar parameter values. It is interesting to note the performance degradation of the SMT solver on the $p = 10, h = 9$ instance of the existential encoding compared to the Boolean encoding.

Summary. It is encouraging to see that the provers have a somewhat complementary capability. Problems solved were in line with predictions made from an understanding of the search styles of the two algorithms: Cooper eliminates variables by considering the values of literals modulo divisibility constraints, this means it is strongly affected by coefficient values and only weakly by Boolean structure (i. e. , depth of formulas, size of disjuncts/conjuncts). The CVC4 solver appears to be based on a projection method (a version of the Omega test), which operates on conjunctions. This requires a conversion to DNF, which can be accomplished more efficiently using a SAT solver working on the propositional abstraction of the LIA formula. The result is that more 'Boolean-like' problems are dealt with efficiently, such as the relational encoded pigeon hole problem, while more 'LIA-like' problems suffer somewhat. This is mitigated by component solvers which allow solving, e. g. , linear systems of equations efficiently.

This suggests that the encoding must be taken into account when using the above theory solvers: 'Boolean-like' encodings will work better when sent to projection style solvers, while arithmetic encodings will work better with Cooper.

3.3.2 Solution Extraction in Cooper's Algorithm

Beagle uses QE in the theory $T_{\mathbb{Z}}$ as a test for satisfiability of BG clauses that have been saturated w. r. t. the calculus derivation rules (i. e. , it tests whether the inference rule Close applies). For most derivations the set of retained BG clauses increases mono-

tonically. Therefore, using a solver that supports *incremental* satisfiability checking would improve performance. An incremental solver is able to use the result of the previous satisfiability check when a new clause is added, perhaps by extending a model of the retained clauses. As mentioned above, there are no known incremental versions of Cooper's algorithm.

This section develops a version of Cooper's algorithm that, for a $\Sigma_{\mathbb{Z}}$-formula $\exists x.\ F[x]$ either returns an integer s such that $F[s]$ is $T_{\mathbb{Z}}$-valid, or concludes that $\exists x.\ F[x]$ is $T_{\mathbb{Z}}$-unsatisfiable. The solution s can be used in the next BG formula generated by the proof procedure, which typically has the form $\exists x.\ (F[x] \wedge G[x])$. The substitution $[x \to s]$ can make the new formula trivially $T_{\mathbb{Z}}$-valid, or at least simplify the formula to $G[s]$. Also, solutions produced from a saturated clause set provide some information about possible counter-examples.

A quick note on semantics: in the following all parameters in $\Sigma_{\mathbb{Z}}$-formulas are replaced with existentially quantified variables. Hence, the only relevant interpretation of $\Sigma_{\mathbb{Z}}$ (without parameters) is the standard model of arithmetic. Similarly, valid usually means $T_{\mathbb{Z}}$-valid and unsatisfiable means $T_{\mathbb{Z}}$-unsatisfiable.

Cooper's algorithm replaces $\exists x.\ F[x]$ with an equivalent finite disjunction (the elimination formula). If the original formula is valid, there must be a subset of valid disjuncts in the elimination formula. After all variables are eliminated from F, the disjuncts of the elimination formula are ground, and so are either $T_{\mathbb{Z}}$-valid or not. Then, for a valid input formula there is always at least one $T_{\mathbb{Z}}$-valid disjunct in the final (ground) elimination formula. The simplest case is where the valid disjunct is in the F_B part of the elimination formula, since each disjunct in F_B is the result of replacing x with some possibly non-ground term. Composing substitutions for each eliminated variable gives a concrete integer value s to substitute for x. It is more complicated where there are no valid disjuncts in the F_B part, however a constraint can be derived by guessing solutions. The following will show how both types of solutions can be extracted during a run of Cooper's algorithm.

It is common to equate formulas over free variables with the set of (tuples of) elements that satisfy the formula in a given model, i. e. , $F[x_1, \ldots, x_n] = \{(x_1, \ldots, x_n) \in D^n \mid T \models F[x_1, \ldots, x_n]\}$. From this perspective, the problem addressed in this section can be phrased: given a valid $\Sigma_{\mathbb{Z}}$-formula $\exists \overline{x}.\ F[\overline{x}]$ find a member of the (non-empty) set described by $F[\overline{x}]$. Moreover, this should be done *during* a run of Cooper's algorithm; simultaneously establishing the validity of the formula and returning a satisfying tuple of integers for the outer existential quantifiers.

A *solution* for x in $F[x]$ is some $\Sigma_{\mathbb{Z}}$-term t not containing x, such that $F[t]$ is equivalent to $\exists x.\ F[x]$. The application of a solution t for x to F is written as for substitutions: $F[x \to t]$ denotes replacing all occurrences of x in F by t. Since t does not contain x, $F[x \to t]$ does not contain x either.

Because of the validity preserving property of solutions they can be safely composed:

Lemma 3.3.1. *If t is a solution for x in $\exists y, x.\ F$ and s is a solution for y in $\exists y.\ F[x \to t]$, then $\exists y, x.\ F \Leftrightarrow F[x \to t][y \to s]$, that is, $[x \to t[y \to s], y \to s]$ is a solution.*

In the following l will be the lcm of all x coefficients in F. This is the same l as used in constructing $\mathsf{unit}_x(F)$ from F. The following lemma shows that to get a correct solution it is necessary to undo the normalization step by removing the factor of l in the solution.

Lemma 3.3.2. *If $[x \to t]$ is a solution for $\mathsf{unit}_x(F)$, then $[x \to t/l]$ is a solution for F.*

Proof. Since $[x \to t]$ is a solution $\mathsf{unit}_x(F)[x \to t]$ is valid. By definition, $\mathsf{unit}_x(F) = F' \wedge l \mid x$, and so $l \mid t$ is valid, implying that t/l is an integer. Given any literal, such as $c(t/l) + s < 0$, in $F[x \to t/l]$ there is a validity preserving transformation back to the literal in F', e. g. , to $t/l + |l/c|s < 0$ by multiplying both sides by $|l/c|$. Hence, each literal in $F[x \to t/l]$ has the same validity status as the corresponding literal in $\mathsf{unit}_x(F)[x \to t]$, and so $[x \to t/l]$ is a solution for F. □

From the presentation in Harrison [Har09], it is clear that solutions to F_B are also solutions to F. Assume $F_B := \bigvee_{j=1}^{D} \bigvee_{b \in B_x} \mathsf{unit}_x(F')[b + j]$ is valid. Then $\mathsf{unit}_x(F')[b + j]$ is valid for some $b \in B_x$ and $j \in [1, D]$. Therefore, $b + j$ is a solution for x in F_B and also to $\mathsf{unit}_x(F')$. By the previous lemma, $b/l + j/l$ is a solution to F, where l is the lcm generated in producing $\mathsf{unit}_x(F)$. Note that b/l is the term $(c_1/l)x_1 + \ldots + (c_n/l)x_n + k/l$ for $b = c_1x_1 + \ldots + c_nx_n + k$

Lemma 3.3.3. *If $\mathsf{unit}_x(F')[b + j]$ is valid for some $b \in B_x$ and $j \in [1, D]$, then $b/l + j/l$ is a solution for x in F.*

If F is valid but F_B is invalid, then it must be that $F_{-\infty}[j]$ is valid for some j. This means that for some sufficiently large and negative x' congruent to j modulo D (the lcm of values d in literals of the form $d \mid x + s$) $F[x']$ is valid. Rather than an exact solution, this produces a constraint on possible solutions.

In order for the solution to produce the same truth values for literals in F as for the corresponding literals in $F_{-\infty}$, the concrete value of x must be below a certain threshold. Specifically, any solution must falsify literals of the form $0 \approx x + t, 0 < x + t$, and must satisfy literals of the form $0 < -x + t, 0 \not\approx x + t$. Construct from $\mathsf{unit}_x(F')$ the set of terms UB by taking:

1. $-t$ if $0 \approx x + t$ or $0 \not\approx x + t$ is in $\mathsf{unit}_x(F')$
2. $-t + 1$ if $0 < x + t$ is in $\mathsf{unit}_x(F')$
3. t if $0 < -x + t$ is in $\mathsf{unit}_x(F')$.

If some element s satisfies $s < t$, then $0 < -s + t$ is valid; if $s < -t + 1$, then $0 < s + t$ is invalid and similarly where $s < -t$. So the literals of $\mathsf{unit}_x(F)[s]$ will have the appropriate sign. Therefore, if $s \equiv j \mod D$ for some j such that $F_{-\infty}[j]$ is valid, then s is a solution for x in $F_{-\infty}$ and in $\mathsf{unit}_x(F')$. As above, if l is the lcm, then s/l is a solution for the original F.

Lemma 3.3.4. *If $F_{-\infty}[j]$ is valid for some j, and s is a domain element which satisfies both $s \equiv j \mod D$ and $s < t$ for all $t \in \mathsf{UB}$, then $[x \to s/l]$ is a solution for F.*

The set of solutions implied by the above constraints is non-empty by construction. However, the set UB can contain non-ground terms, meaning that it may not be possible to evaluate the constraint until other variables are eliminated. Once all terms in UB are ground, the constraint can be solved by finding the least n such that $j - nD$ is less than every term in UB.

A similar analysis also works for the dual transformation, using F_∞ ($F_{-\infty}$ with replacements reversed) and F_B, where B consists of upper bounds for x.

3.3.2.1 Constructing Solutions

Solutions are constructed by building the elimination formula for just a single disjunct at a time, rather than the formula as a whole. For example, given $\exists x \exists y.\ F[x, y]$, y is eliminated producing $\exists x.\ G_1 \vee G_2 \vee G_3$ say, then x is eliminated from each of $\exists x.\ G_i$ individually. Effectively, this is a depth-first search of all possible disjuncts, where each disjunct (state) has a chain of selected solutions for the variables previously eliminated. Clearly, once all variables are eliminated, the final disjunct is either valid or invalid and, if valid, all of the solutions associated with the state have ground constraints that can be evaluated.

The following algorithm implements such a search, returning either a solution or $\bot$ where the input is not valid.

Let F be a $\Sigma_{\mathbb{Z}}$-formula, xs a list of variables to be eliminated and σ an existing solution which may be empty. It is assumed that all free variables in F appear in xs. The cooper sub-procedure expands a formula to the equivalent elimination formula for the given variable. The $\mathrm{complete}$ sub-procedure fills in the parameteric values in a partial symbolic solution, justified by Lemma 3.3.6.

Solutions are constructed by collecting sets of constraints on solutions and evaluating them as more quantifiers are eliminated. These constraints are called *symbolic solutions* to emphasize that they are possibly non-ground representations of solutions. There are the two types of symbolic solutions possible for a variable: assignment solutions and bound solutions, written assign and bound in the code above.

A symbolic solution is *closed* if it has no variables in t, for $\mathsf{assign}(t)$, and no variables in any term in UB for bound. Closed symbolic solutions can always be replaced by a solution (an assignment to a concrete integer).

Example 3.3.2. Using $F_{-\infty}$ and F_B, as in Example 3.3.1. Every disjunct of $F_{-\infty}$ is invalid except where $j = 3$. So the symbolic solution for x is $\mathsf{bound}(3, 3, \{y\})$.

Next eliminate y from $G = 0 \not\approx y - 5$.

$$\begin{aligned} \mathsf{unit}_y(G) &= G \\ l_y &= 1 \\ G_B &= 0 \not\approx (5+1) - 5, \\ &\quad \text{where } B_y = \{5\}, b = 5 \text{ and } j = 1 \\ G_{-\infty} &= \top \end{aligned}$$

```
1  algorithm getSymbolicSolution(F,xs,σ):
2   if xs.isEmpty:
3    if F = ⊤ return σ
4    else return ⊥
5
6   foreach disjunct D in cooper(F, xs):
7    if D = F_B[b + j]:
8     let solution := σ · [x->assign(b+j/l)]
9    else if D = F_−∞[j]:
10    let solution := σ · [x->bound(j,D,UB(unit(F)))]
11
12   if D = ⊤:
13    return complete(solution, xs)
14   else if D != ⊥:
15    let s := getSymbolicSolution(D, y, solution)
16    if s != ⊥:
17     return complete(s, xs)
18
19  //nothing is valid
20  return ⊥
```

Figure 3.4: Creates the symbolic solution resulting from eliminating all of xs from F.

There is one valid disjunct in G_B, and this gives the symbolic solution for y: $\mathsf{assign}((5+1)/1)$. After applying the solution for y, the symbolic solution $\mathsf{bound}(3, 3, \{6\})$ for x is closed and can be evaluated to $x = 3$. Since l_x was 3, the final solution is $x = 1$. So the solution for F is $\{x \to 1, y \to 6\}$ and, as a final check, $0 < -3 \cdot 1 + 6 \wedge 0 \not\approx 6 - 5$ is valid. □

Lemma 3.3.5. *F has a closed symbolic solution iff it is valid*

Proof. By Lemmas 3.3.4 and 3.3.3, solutions to closed symbolic solutions are solutions to F. Conversely, a valid F has at least one valid disjunct in its elimination formula. The algorithm $\mathrm{getSymbolicSolution}$ must eventually find it, as the final elimination formula for F has a finite number of disjuncts. □

The algorithm in Figure 3.3.2.1 can terminate before eliminating all variables in `xs` and return a symbolic solution. This can happen when a disjunct contains a subset of `xs`. The result is not a closed solution for F, as the remaining variables are not assigned values. The following lemma shows that free variables in a symbolic solution returned by $\mathrm{getSymbolicSolution}$ can be filled in arbitrarily and still yield a solution for F. In the code this is done by the call to sub-procedure $\mathrm{complete}$ on lines 13 and 16.

Lemma 3.3.6. *A symbolic solution for F can always be evaluated to a solution α for F.*

Proof. There are two cases: either all symbolic solutions are closed or some are open. When all solutions are closed, the solution can be evaluated (i. e. , solutions to bound

constraints can be found using Lemma 3.3.4). When a solution is open, then that solution depends on variables that were not eliminated. This happens when elimination produces, e. g. , $F \Leftrightarrow \exists y.\ F'[y] \vee \top$. It must be shown that using an arbitrary solution for variables in open solutions does not affect the validity of F. Assume variables $\gamma = \{x_1, \ldots, x_k\}$ are not eliminated from $\text{vars}(F) = \{y_1, \ldots, y_l, x_1, \ldots, x_k\}$ and that there is a symbolic solution for each variable y_i. The Cooper expansion of $\exists y_1, \ldots, y_l, x_1, \ldots, x_k.\ F$ is $\exists x_1, \ldots, x_k.\ \top$, since a solution exists. Let $[x_1 \to c_1, \ldots, x_k \to c_k]$ be a hypothetical solution for arbitrary integers c_i. A representation of this solution can be added to the original F like so:

$$\exists y_1, \ldots, y_l, x_1, \ldots, x_k.\ (F \wedge x_1 \approx c_1 \wedge \ldots \wedge x_k \approx c_k)$$

Since no variable in γ is eliminated in the Cooper expansion of this formula, the result of the elimination procedure is $\exists x_1, \ldots, x_k.\ (\top \wedge x_1 \approx c_1 \wedge \ldots \wedge x_k \approx c_k)$, which remains valid. Therefore, extending a symbolic solution for a valid formula with arbitrary values for open solutions produces a valid closed solution. □

For example, $x + y \approx 0$ has $B_x = \{-(y+1)\}$ and symbolic solution $\sigma = [x \to -y, y \to y]$. The corresponding disjunct is $\exists y.\ 0 \approx 0$ (substitution with $b = -(y+1), j = 1$). Adding a guessed solution $y = c$, $\exists x, y.\ (x + y \approx 0 \wedge y \approx c)$, results in the same B_x and the same σ. However, the final formula is $\exists y.\ (0 \approx 0 \wedge y \approx c)$.

3.3.2.2 Performance of Caching in Beagle

The solution extraction method is implemented in *Beagle* as above. It is integrated in the main proof search in a straightforward way: each time a BG clause is retained the solver checks for a solution. (This is the same place the Cooper solver is usually called). If there is a solution, then it is applied to the entire set of retained BG clauses, and these are simplified. If any variables remain in the simplified clauses, then the remaining variables are eliminated using the solution extraction method. This yields either a new solution, or a 'false' result. In the latter case, or if the application of the stored solution produces a 'false' result, then the algorithm is restarted with an empty solution.

When a non-deterministic split occurs, the current solution is stored with the decision level, before entering a new decision level. When a split is backtracked to, the corresponding solution from that decision level is reinstated.

As described below, the solution extraction algorithm has some built-in inefficiencies which may or may not balance the advantages offered by caching.

In tests on arithmetic (also including uninterpreted symbols) problems from the TPTP-v6.4.0 library, just 27 problems were solved significantly faster. Two of these: `ARI659=1` and `SWW619=1`, were not solved when caching was disabled. However, 97 problems suffered a greater than 1s performance reduction (although most had less than 10s degradation) and 577 saw no change at all. Problems which showed any improvement were 4s faster on average, while those that showed performance reduction were 8s slower on average. This is indicated by Figure 3.3.2.2.

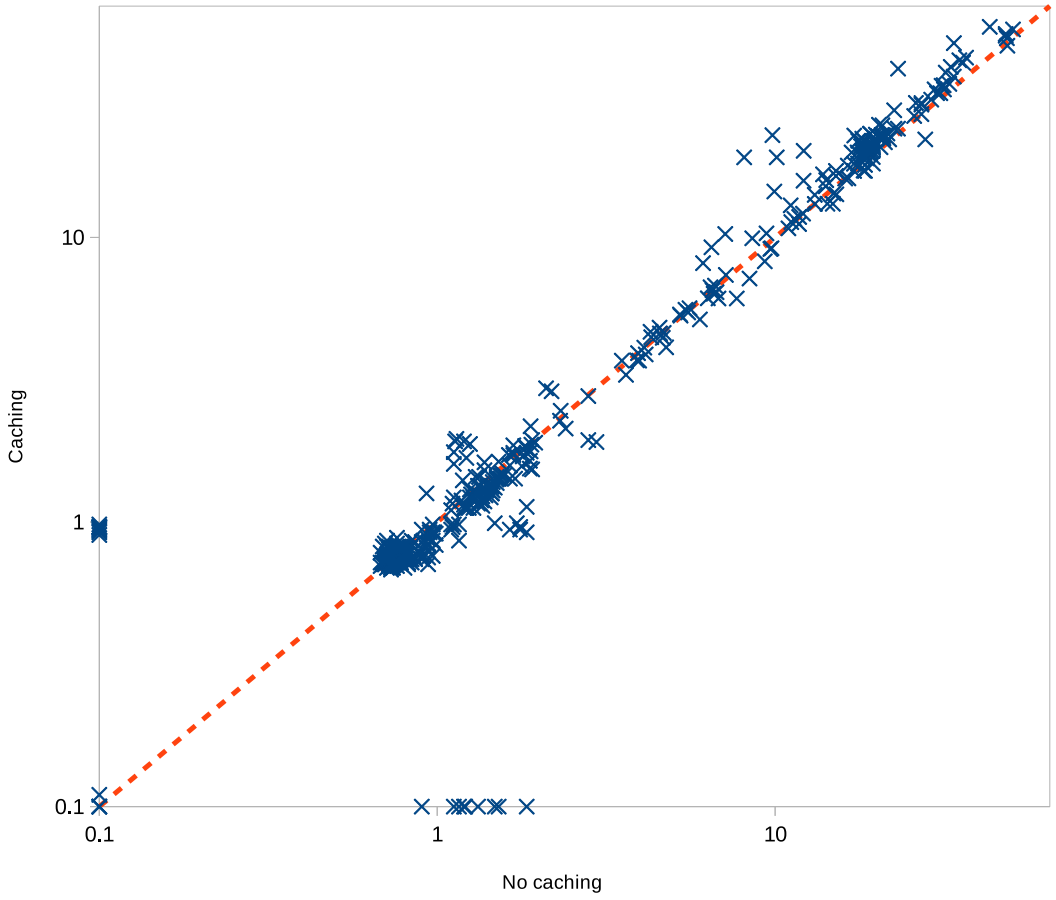

Figure 3.5: Run time in seconds of *Beagle* with and without Cooper solution caching

In general, most arithmetic problems in the TPTP are discharged quickly by Cooper without requiring 'deep' arithmetic reasoning. It seems that the accumulated inefficiencies of many small calls do not counter the advantage of stored solutions, except in a few cases.

Future work Monniaux [Mon10] divides arithmetic quantifier elimination procedures into two classes: those which, like Cooper, substitute an infinite disjunction $\exists x.\ F[x]$ for a finite disjunction $\bigvee_i F[x \to x_i]$, where x_i are terms over the free variables of F; and those which project conjunctions of atoms, e. g. , Fourier-Motzkin for LRA and the Omega test for LIA. There are QE methods similar to Cooper's for Rationals and Reals, e. g. , Ferrant and Rackoff's, or Loos and Weispfenning's for Reals. Perhaps a similar analysis could be applied to obtain solution caching methods for those theories too.

There is a collection of enhancements made to the standard Cooper implementation which have not been applied to this version yet. These enhancements either short circuit the usual Cooper expansion, e. g. , elimination of equations, or reorganize it internally, e. g. , multiplication of disjuncts after a variable elimination. The short-circuiting enhancements could easily be added as a filter before using solution-extracting Cooper.

The algorithm given above is less efficient than the usual Cooper algorithm because it specifically avoids the case where F is proven valid by finding a non-singleton set of valid disjuncts, e. g. , $(x \approx t) \vee (x \not\approx t)$. The intuition behind choosing only disjuncts in the F_B subformula of the elimination formula was that more specific solutions would apply to more formulas later on, though this is not necessarily the case. It could be possible to have a version of Cooper's algorithm which simply records the *sets* of symbolic solutions and chooses a representative one once a valid subformula is found, rather than using the depth-first search given here.

Otherwise, the solution extracting version could be used asynchronously: when the usual Cooper algorithm reports a formula valid, the solution extracting version can be called to find a solution while the proof search continues. As soon as a solution is found, it can be tried, working on the assumption that solutions for subsets of BG clauses are likely to be solutions for the whole set. However, if BG clauses are generated too quickly, then each solution may end up being discarded, with no positive effect at all.

Stored solutions to the inner quantifier in an unsatisfiable EA-formula may be seen as witnesses to unsatisfiability of the overall formula. Specifically, a stored solution after elimination of y in $\exists x \forall y.\ \phi[x, y]$ is a map $m(x)$ such that $\forall x.\ \neg\phi[x, m[x]]$ is valid. In the lucky case that some bounds formula disjunct is shown valid immediately after elimination of y, then that map is simply a linear polynomial term. For example, elimination of y from $\exists x \forall y.\ y > x$ yields a bounds formula $0 \approx 0 \vee 0 < 0$ after substituting $y = x$, and clearly that fulfils the role of m above. The case where validity is not immediate or follows from the $\phi_{-\infty}$ formula is less clear.

3.4 Proof Procedure

This section provides a summary of *Beagle*'s proof procedure. The proof procedure follows standard techniques, but treats BG formulas separately on some occasions.

Preprocessing. *Beagle* accepts input formulas in two alternate syntaxes, TPTP-TFF [SSCB12] and SMT-LIB version 2.0 [BST10]. The SMT-LIB language is richer than the TPTP-TFF language due to its support for polymorphic sorts and functions. The SMT-LIB also features predefined theories such as arrays and lists as described in previous chapters. *Beagle* automatically monomorphizes sorts and function symbols, and it includes data structure theory axioms as needed when processing SMT-LIB files.

Both TPTP-TFF and SMT-LIB provide syntax for full first-order logic (not just clausal logic). *Beagle* has two translators into clause normal form (CNF), a standard one and a Tseitin-style translator which introduces definitions for 'complex' subformulas. The default is the standard CNF translator, because it gave better results overall on the problems in the TPTP. However, the Tseitin transformed CNF was needed for many SMT-LIB problems, to support the `let` keyword and to reduce the size of large ground problems.

CNF transformation includes Skolemization of existentially quantified variables and treats existentially quantified integer variables in a special way, by removing them with QE instead of Skolemization, if possible. For example, the input formula $\forall x : \mathbb{Z}.\ p(x) \vee \exists y : \mathbb{Z}.\ y \not\approx x + 1$ becomes $\forall x : \mathbb{Z}.\ p(x)$, whereas Skolemization would have given $\forall x : \mathbb{Z}.\ p(x) \vee \mathsf{f}(x) \not\approx x + 1$. In particular, if the input formulas are all BG formulas over the integers, no Skolem functions are introduced, and so *Beagle* is a decision procedure for that class.

Main loop and simplification. *Beagle*'s main loop is the well-known 'Discount loop'. It maintains two clause sets: Old and New. Old is initially empty, while New is initialized with the input clauses. At each iteration, a *selected* clause is removed from New and simplified using clauses from Old and New. The simplified clause is then added to Old and all possible inferences between it and clauses in Old are performed. The resulting clauses are simplified by clauses in Old and added back to New again, closing the loop. If any result is a BG clause, the BG solver is called with the new set of BG clauses. Lemmas are treated specially and are added to Old at the beginning of a derivation. This allows only inferences and simplifications between lemma clauses and input clauses, never between lemma clauses.

Simplification techniques include standard ones: demodulation by unit clauses, proper subsumption deletion, and removal of positive literals L from a clause in the presence of a unit clause that instantiates to the complement of L. All clauses in Old are mutually simplified, and backward simplification is optional.

By default, a split rule is enabled that breaks clauses into variable-disjoint subclauses and branches out correspondingly. Dependency-directed backtracking (Figure 3.2.2) is used to avoid exploring irrelevant cases.

The default term ordering is LPO if BG theories are present, otherwise it is KBO. See Baumgartner and Waldmann [BW13b] for properties of the LPO specific to Hierarchic Superposition.

Fairness. Fairness is achieved by a combination of clause weights and clause derivation age. It can be tuned by setting the 'weight-age-ratio' parameter, a non-negative number relating how many lightest clauses are selected before the oldest clause is selected. Clause weights are computed in such a way that selection by weight alone would be a fair strategy. The default weight-age-ratio value is five.

Auto mode. *Beagle* includes a simple auto mode that switches between several strategies. When on, *Beagle* first tries the default flag setting. If there is no result within one third of the given time limit, *Beagle* restarts using aggressive simplification and unabstraction settings. As described above, these are incomplete, though a contradiction derived using such rules remains valid. If no contradiction is derived after two-thirds of the allocated time has elapsed, the final strategy is used whereby BG variables in the input may be instantiated with BG-sorted FG terms rather than only with BG terms. Specifically, all abstraction variables in the input are replaced by general variables. As previously mentioned, this is 'more complete' but creates a larger search space. No part of the previous run's state is kept between restarts.

Proof Output Successful proofs can be output as a TFF formatted[7] derivation of either an empty clause or an unsatisfiable set of BG clauses. This can be used to interface with other tools that support the format, such as the Isabelle proof assistant. Presently, CNF transformation and refutation of BG clause sets is not documented in the derivation, though some BG simplifications are described. Derivations are reconstructed by keeping a derivation record with each clause which points to the premise clauses in an inference that derived the clause.

3.4.1 Implementation

Beagle implements support for both the TPTP-TFF and SMT-LIB input languages using Scala's parser combinator library. *Beagle*'s internal formula representation follows TFF, so to support the SMT-LIB standard it must perform sort monomorphization and add axioms for predefined theories like ARRAY. Parsing of SMT-lib files is done with the help of the separate SMTtoTPTP library [Bau15]. Also, *Beagle* includes an implementation of a sort un-erasure algorithm [CS03], which lifts unsorted first-order formulas to many-sorted logic. This may improve performance on unsorted first-order problems, but as this translation is incomplete in general, it is disabled by default.

Beagle uses a simple term-indexing scheme which is essentially top-symbol hashing. This is used to retrieve term positions eligible for superposition or demodulation

[7] Format described at http://wwww.tptp.org

within clauses. Discrimination-tree indexing is used for forward simplification, including both demodulation and subsumption by unit clauses.

Scala specific features. *Beagle* makes heavy use of many built in Scala data structures, primarily `List`, `Vector` and `Map`. Not only are the implementations well optimised, but they also provide powerful abstractions allowing for simple and maintainable code.

Scala's declarative style encourages the use of immutable values, which minimizes data duplication. Scala also provides a lazy evaluation feature, which is extremely useful for caching data: e. g. , the computation of maximal literals in a clause can be deferred until the clause becomes eligible for an inference, it may never be computed if the clause is simplified first. The Scala REPL interpreter is an invaluable tool for debugging: for example, one could take the (usually large) result of an invalid derivation and programmatically investigate it using functional operators like map or filter.

The simple structure of logic formulas and clauses is a good fit for property based testing, using libraries such as `scalacheck`[8], which use grammars to generate random test data. These data are used as input for properties given as universally quantified predicates.

3.5 Performance

3.5.1 TPTP

This section reports results of running *Beagle* on the first-order problems from the TPTP–v6.4.0 problem library [Sut09] that involve some form of arithmetic, including non-linear, rational and real arithmetics.

The experiments were carried out on a Linux desktop with a quad-core Intel i7 cpu running at 2.8 GHz, with 8GB of RAM, although the host JVM[9] was configured with maximum heap size of 4GB. The CPU time limit was 60 seconds soft (solver's heuristic target time) and 65 seconds hard (unresponsive processes killed).

Of 1161 total problems, *Beagle* returned the correct result on 869 problems, within the time limit. For some problems *Beagle* produced a saturated clause set (saturated under the inference rules of the Hierarchic Superposition calculus), but was not able to conclude $\mathcal{B}$-satisfiability due to the presence of free BG-sorted operators[10]. Table 3.2 summarizes the global statistics for this test.

The automatic strategy selection heuristic was used for this run. After a predetermined time, this restarts the proof using incomplete (strong) simplification and later, weakens restrictions on variable substitutions by setting all BG-sorted variables in the input clause set to be 'general' variables. In total, the default strategy was used

[8] http://scalacheck.org/

[9] OpenJDK v.1.8

[10] Non-linear multiplication is included in this category for this experiment.

	Solved	Unknown	Timeout	Total
Theorem	835	16	221	1072
Non-theorem	34	24	30	88
Total	869	40	251	1161

Table 3.2: TPTP statistics

875 times, the second strategy 4 times (3 theorems found), and the last strategy 37 times (5 theorems found, 30 'unknown' results).

Table 3.3 summarizes the results per category, and Table 3.4 displays the same results against their TPTP difficulty rating. In the TPTP problem library, problem ratings are given as a real number between 0 and 1. Problems receive a rating of 0 if all theorem provers (specifically all systems entered in the relevant category of the most recent CASC) can solve it, and a rating of 1 if none of those solvers can solve it.

By TPTP problem category, *Beagle*'s best performance was on ARI, DAT, GEG and NUM. These are characterized by smaller problem sizes with an arithmetic reasoning component. GEG problems were largely solved by optimizing simplification inside the Cooper solver. On the other hand, performance was much worse on those problems which involve large problem sizes, specifically HWV problems (large EPR encodings of bounded model-checking). This is due to the size of the formulas and emphasis on boolean reasoning. A typical trace from an HWV problem shows all of the time spent performing superposition inferences (technically the inferences are simple resolution inferences) and simplifications via subsumption. It is quite likely that well-known enhancements like feature vector indexing [Sch13] for subsumption and hyper-resolution [Rob65a] would significantly improve performance for this problem class.

The remaining easy (rated < 0.1) problems that *Beagle* failed to solve involved multiplication operators and several HWV problems.

The three solvable problems with a rating of 1.0 are ARI635=1, ARI636=1 and ARI633=1.

The same problems were tested using Z3 [dMB08] as the background solver for each of the integer, rational and real theories. Using Z3 *Beagle* was able to solve just one additional problem: SWW609=2. Overall, *Beagle* with Z3 performed better on 60 problems and worse on 12 problems, with an average time difference of 0.15s faster than the default solver. The problems on which Z3 performed better ranged in difficulty from 0 to a maximum of 0.57, with the majority of the improvement found for problems with difficulty 0.14. Table 3.5 shows these distrubtions. Each row counts problems which showed at least a 1s performance improvement in favour of the respective BG solver configuration. The top row lists difficulty ratings for those problems.

It is possible that this performance could be improved by using a programmatic inferface to the SMT solver, or by fine-tuning the settings that are used for the SMT solver. Previous results showed little difference in performance whether case based reasoning (i. e. , applications of the `Split` rule) on BG clauses was carried out by the

	Theorem		Non-Theorem	
Category	Total	Solved	Total	Solved
ARI	642	572	25	16
DAT	100	95	5	0
GEG	5	5	0	0
HWV	88	1	12	0
MSC	3	3	0	0
NUM	42	41	24	17
PUZ	1	1	0	0
SEV	4	2	2	0
SWV	2	2	2	0
SWW	181	111	18	0
SYN	1	0	0	0
SYO	3	2	1	1
Total	1072	835	88	34

Table 3.3: *Beagle* performance on the TPTP arithmetic problems by category.

Rating	0	0.14	0.17	0.29	0.33	0.43	0.5	0.57	0.67	0.71	0.83	0.86	1
Total	521	266	17	32	22	50	21	35	33	47	3	30	84
Solved	511	262	15	25	5	24	12	8	4	1	0	1	3

Table 3.4: *Beagle* performance on the TPTP arithmetic problems by problem rating.

foreground solver or the background solver and this applied to both Cooper and Z3.

To put *Beagle*'s performance in context with similar automated reasoning tools, we reproduce here the results of the CASC-J8 competition. In the competition, the provers were run on standardised hardware with preset configurations given by their designers. Full competitions details are available in the proceedings [Sut16]. Problems in the Typed first-order division reported in Table 3.6 were drawn from TPTP-v6.4.0 (as for previous result tables), consisting of just theorems. Solvers can optionally provide a proof of the hypothesis (the solutions row) and 'New Solved' refers to problems which were not publically available prior to the competition.

Table 3.7 shows results on arithmetic problems which are counter-satisfiable. In this case a solution describes a model for the counter-example; a difficult problem given that these often have infinite domains. The asterisk in the first result row

	0	0.14	0.29	0.43	0.5	0.57
Z3	5	43	6	2	1	3
Default	7			4		1

Table 3.5: Performance distribution (count of problems solved in faster time) for different BG solver configurations

System	Vampire	VampireZ3	CVC4	Beagle	Princess
Version	4.1	1.0	TFF-1.5.1	0.9.47	160606
Solved	419/500	380/500	343/500	300/500	342/500
Av. CPU Time (s)	13.39	9.15	5.72	18.76	17.59
Solutions	419/500	380/500	343/500	300/500	271/500
New Solved	3/6	3/6	0/6	5/6	5/6

Table 3.6: CASC-J8 Typed First-order theorem division.

System	CVC4	Beagle	CVC4	Princess
Version	TFN-1.5	SAT-0.9.47	TFN-1.5.1	160606
Solved	10/50*	10/50	9/50	8/50
Av. CPU Time (s)	32.27	3.11	0.02	1.44
Solutions	0/50	0/50	9/50	0/50
New Solved	1/7	0/7	2/7	0/7

Table 3.7: CASC-J8 Typed First-order non-theorem division.

indicates that the results reported here differ from those available online [11], as it was later discovered that there was a bug in that version of CVC4. *Beagle* then won the category with a lower average time per problem.

3.5.2 SMT-LIB

This section reports the performance of *Beagle* [12] on the 2014 release of SMT-LIB benchmarks[13] focusing on the logics with an arithmetic component. Specifically these were ALIA, AUFLIA, UFLIA, UF_IDL (integer difference logic) and the corresponding quantifier-free problem sets, including QF_LIA. (The LIA category was ignored as it contains only problems from the TPTP). Only those problems indicated as unsatisfiable in the problem description were selected. *Beagle* was run with automatic strategy selection (as described above). We found a mix of results: *Beagle* was able to solve a few problems unsolved by SMT solvers[14] yet there were also quite a few problems that were marked as 'trivial' (all SMT solvers in the SMT-Eval 2013 can solve them in under five seconds), which *Beagle* could not solve. Table 3.5.2 describes problems solved by category, where QF refers to the quantifier-free fragment of the logic to its left.

In total *Beagle* solved 89 problems not solved by SMT solvers. Those problems are summarized in Table 3.5.2, the listed categories are subcategories of 'UFLIA/sledgehammer':

There were many problems which *Beagle* could not parse, as it is not optimized for large problem sets. In total there were 1,391 trivial problems not solved by *Beagle* .

[11] http://www.cs.miami.edu/ tptp/CASC/J8/WWWFiles/ResultsSummary.html
[12] version 0.9.25
[13] http://smtlib.cs.uiowa.edu/benchmarks.shtml
[14] For this we used the difficulty ratings given for SMT-Comp 2014.

Logic	ALIA	QF	AUFLIA	QF	UFLIA	QF	UFIDL	QF	QF_IDL	QF_LIA
Total	41	72	4	516	6602	195	62	335	694	2610
Solved	31	40	4	205	1736	155	42	29	24	28

Table 3.8: SMT-lib theorems solved by category.

Category	Arrow_Order	FFT	FTA	Hoare	StrongNorm	TwoSquares
Solved	17	2	34	20	2	14

Table 3.9: Difficult SMT-lib theorems and their categories.

It was not possible to draw broad conclusions about which categories *Beagle* is best suited to. For example, all of the hardest problems *Beagle* solved were among the UFLIA benchmarks, but there were also at least 200 trivial problems from that category were unsolved (in the 'simplify' and 'simplify2' subcategories). Also it was hypothesised that *Beagle* would perform much worse in the quantifier-free fragment, and that was the case for QF_IDL and QF_LIA, but not so for QF_UFLIA and QF_AUFLIA.

3.5.3 CADE ATP System Competition (CASC)

Beagle was a regular participant in the annual CASC event, in which provers compete to solve randomly selected TPTP benchmarks. The benchmarks are divided into categories such as arithmetic, pure first-order, large-theory base, and non-theorems. Provers are ranked based on number of problems solved, whether a proof was output, and general efficiency. This section summarizes results from the three most recent events from oldest to newest (CASC-J8).

CASC-J7. [Sut15] *Beagle* was entered in the TFA division (Typed First-order Arithmetic theorems). For this division, the problem set consists of typed first-order problems with an arithmetic component over integers, rationals, or reals, of which roughly half were previously unseen by competitors.

Other solvers entered in the TFA category were *CVC4* [BCD+11], *SPASS+T* [WP06], *Zipperposition* [Sut15], and *Princess* [Rüm08]. In terms of overall problems solved, *Beagle* placed third equal with 173/200 solutions, only three fewer than the winning solver *CVC4*. *Beagle* performed quite well in terms of mean efficiency (solutions per second multiplied by number of solutions); it was outperformed by only *CVC4* [15].

CASC-25. [SU16] This saw the introduction of the TFN (Typed First-order arithmetic Non-theorem) division. *Beagle* solved 6 of 20 problems in that division, coming second equal. The winning solver, CVC4, solved just 10. *Beagle* had no specific

[15] For an explanation of how mean efficiency is computed see the CASC-J7 proceedings [Sut15].

strategy for solving such problems, simply relying on saturation via Superposition. Soundness issues due to a lack of sufficient completeness were avoided by reporting 'unknown' for problems with free BG sorted symbols.

In the TFA division, *Beagle* solved 131 of 200 problems, the lead was taken by the improved *Vampire* solver [KV13], which previously dominated the pure first-order division. As observed above, *Beagle* struggled to solve larger problem instances with involved boolean reasoning, as found in the SWW and HWV problem sets.

CASC-J8. [Sut16] Both TFA and TFN categories were expanded to 500 and 50 problems respectively. Using the negation disproof strategy for multiplication solving as described in Chapter 4 allowed *Beagle* to win the TFN division, solving 10 of 50 problems. *CVC4* was a close second, solving 9 problems.

In the TFA category *Beagle* solved 300 problems, coming in fourth place. A major help was the inclusion of a comprehensive set of lemmas for BG reasoning (see Section 3.2.1), as well as a partial instantiation heuristic for finite sorts. This helped to eliminate many over-productive clauses from the input. Although returning fewer solutions, *Beagle* outperformed *Princess* by returning more proofs. Again *Vampire* took the lead, with 419 solutions; a testament to years of research and implementation improvement.

3.6 Summary

Beagle implements the Hierarchic Superposition calculus with weak abstraction, an enhancement that sets it apart from other implementations of the *HSP* calculus, such as SPASS(LA). It also has an optimized implementation of Cooper's algorithm for quantifier elimination in $T_{\mathbb{Z}}$, and uses an off-the-shelf Simplex solver for reasoning in $T_{\mathbb{Q}}$ and $T_{\mathbb{R}}$. The capability for fast reasoning on $\Sigma_{\mathbb{Z}}$-formulas with multiple quantifiers is exploited to allow the inclusion of parameters in the BG theory, and pure $\Sigma_{\mathbb{Z}}$-formulas can be discharged without invoking the Superposition procedure at all.

A method for extracting example values for existentially quantified variables in satisfiable $\Sigma_{\mathbb{Z}}$-formulas was given. Though it did not improve performance when used to generate cached solutions during a proof search, it remains a useful capability.

The performance of the Cooper solver was measured on several classes of problems parameterized both in the number of variables and number of quantifier alternations. For one problem class, a state of the art SMT solver was unable to solve any instances, while the Cooper solver could. Also, two different encodings of the pigeon-hole problem illustrate a sensitivity to 'Boolean-like' encodings: the SMT solver performed relatively better on the latter, while Cooper performed better on an encoding that used the inherent structure of $T_{\mathbb{Z}}$.

Beagle's performance on the latest version of the TPTP was given, as well as reports from recent year's CASC events. In the latest, *Beagle* won the typed non-theorem category using a technique suggested in Chapter 4.

3.6.1 Availability

Beagle is available at https://bitbucket.org/peba123/beagle under a GNU General Public license. The distribution includes the Scala source code and a ready-to-run Java jar-file.

Chapter 4

Definitions for Disproving

4.1 Motivation

This chapter addresses the problem of automatically disproving invalid conjectures, CON, over data structures such as lists and arrays over integers (axiomatized by AX), in the presence of additional hypotheses, HYP, over these data structures. Such invalid conjectures come up frequently in applications of automated reasoning to software verification or in goals produced by interactive theorem provers.

The disproving problem is to show that $\mathsf{AX} \cup \mathsf{HYP}$ does not entail a sentence CON. The obvious approach to disproving is to show satisfiability of $\mathsf{AX} \cup \mathsf{HYP} \cup \{\neg\mathsf{CON}\}$ by means of a complete theorem prover. Unfortunately, current theorem proving technology is of limited usefulness for disproving: finite model finders cannot be used because the list axioms do not admit finite models; SMT-solvers are typically incomplete on quantified formulas and face the same problem; and theorem provers based on saturation often do not terminate on satisfiable input, and are incomplete when background theories are present.

Nevertheless, *refutation* complete theorem provers should be able to tackle the case where CON is *contradictory* with $\mathsf{AX} \cup \mathsf{HYP}$, rather than simply non-entailed. In that case the set $\mathsf{AX} \cup \mathsf{HYP} \cup \{\mathsf{CON}\}$ is unsatisfiable. The usual application of refutation complete provers concludes $\mathsf{AX} \cup \mathsf{HYP} \models \mathsf{CON}$ when deriving a contradiction from $\mathsf{AX} \cup \mathsf{HYP} \cup \{\neg\mathsf{CON}\}$. Similarly, the situation described above proves that $\mathsf{AX} \cup \mathsf{HYP} \models \neg\mathsf{CON}$. But this does not mean that $\mathsf{AX} \cup \mathsf{HYP} \not\models \mathsf{CON}$, since we assume, pessimistically, that only AX is satisfiable a priori. It may be that $\mathsf{AX} \cup \mathsf{HYP}$ is itself unsatisfiable, and so CON is entailed *ex falso*. In summary, to show $\mathsf{AX} \cup \mathsf{HYP} \not\models \mathsf{CON}$ requires showing both that $\mathsf{AX} \cup \mathsf{HYP} \models \neg\mathsf{CON}$ and that $\mathsf{AX} \cup \mathsf{HYP}$ is satisfiable.

The specific approach to be described consists of first assuming that AX is satisfiable, then providing templates for HYP that are guaranteed to preserve satisfiability of $\mathsf{AX} \cup \mathsf{HYP}$. Disproving is attempted simply by proving that $\mathsf{AX} \cup \mathsf{HYP}$ entails $\neg\mathsf{CON}$, i. e. , that $\mathsf{AX} \cup \mathsf{HYP} \cup \{\mathsf{CON}\}$ is unsatisfiable.

Section 4.2 gives a general characterization of satisfiability preserving formulas HYP, called *admissible definitions*, and introduces a simple classes of formulas that are admissible. These classes do not include recursive functions however, excluding

many interesting examples. Section 4.3 provides a way around this by giving a fixed syntactic template for some recursive functions, especially those over lists. Templates are given for both predicate and function definitions (which require an extra check) as well as a method inspired by functional programming for instantiating higher-order T_{LIST} functions to produce new admissible functions without resorting to mechanical checks. Finally, Section 4.4 describes some uses and occurrences of those syntactic patterns in applications, as well as giving a practical demonstration that the method of proving non-entailment increases the set of problems which can be solved using both saturation-based reasoners and SMT solvers.

4.1.1 Assumed Definitions

This chapter deals with formulas rather than clauses, so semantics using valuations is assumed (see Section 2.2 in Chapter 2). There is no division of the signature into hierarchic specifications, however, all signatures including integers must have $T_{\mathbb{Z}}$-extending interpretations.

Tuples of terms are written $\bar{s}$, and $\overline{\forall}$ denotes universal closure.

Recall that $\Sigma \cup \{f\}$ denotes the addition of the operator f to the signature Σ. As this is short for $(\Xi_\Sigma, \Omega_\Sigma \cup \{f\})$, it is implicit that the arity of f is over sorts in Ξ_Σ. A Σ-interpretation is *extended* to a $(\Sigma \cup \{f\})$-interpretation by the addition of an interpretation for f; the domain of the new interpretation is the same as that of the existing interpretation. A Σ-interpretation is extended *by a formula* ψ_f if it can be extended to a $(\Sigma \cup \{f\})$-interpretation that satisfies ψ_f, again, keeping the same domain.

The theory $T^{=}_{ARRAY}$ is used in the examples in Section 4.4, it includes an extra operator $\mathsf{init} : \mathbb{Z} \mapsto \mathsf{ARRAY}$, defined by the extra axiom:

$$\mathsf{read}(\mathsf{init}(x), i) \approx x$$

So a term $\mathsf{init}(t)$ represents an array that is initialized everywhere with t.

The satisfiability of the list axioms is well known and can be determined automatically using a Superposition based calculus [ABRS09]. Using hierarchic specifications, for example when using integers form the element theory, the theorem prover *Beagle* [BW13b] in a complete setting and after adding the axioms $\exists d_{\mathbb{Z}}.\ \mathsf{head}(\mathsf{nil}) \approx d$ and $\mathsf{tail}(\mathsf{nil}) \approx \mathsf{nil}$, will terminate on $\mathsf{AX}_{\mathsf{LIST}}$. Because the axioms have sufficient completeness (see Chapter 6), there is a model for $\mathsf{AX}_{\mathsf{LIST}}$.

4.2 Admissible Definitions

In typical applications, HYP consists of *definitions* for new operators which extend the signature of AX, and also appear in the conjecture. New functions f and predicates p are defined relative to an existing signature Σ that does not contain either f or p,

using first-order formulas with the following forms:

$$\forall \overline{x}, y.\ f(\overline{x}) \approx y \Leftrightarrow \phi_f[\overline{x}, y] \tag{4.1}$$

$$\forall \overline{x}.\ p(\overline{x}) \Leftrightarrow \phi_p[\overline{x}] \tag{4.2}$$

where ϕ_f and ϕ_p are Σ-formulas. It is clear already that any extension with a predicate p and definition formula (4.2) is immediately satisfiable; the new predicate is just an alias for a formula that is completely specified in the existing interpretation. Since p and f are not present in Σ, it is not possible to include contradictory formula such as $p(x) \approx \mathit{true}$ and $p(x) \not\approx \mathit{true}$ in ϕ_p, for example.

Example 4.2.1 (Extensional Set Definitions)**.** An extensional set definition describes a set by listing exactly the elements it contains, e. g. $S = \{a, b, c\}$ is described by $\psi[x] = x \approx a \vee x \approx b \vee x \approx c$. The set can be named by introducing a new symbol: $\forall x.\ s(x) \Leftrightarrow \psi[x]$, and this is in form (4.2) above, assuming all of a, b, c are in Σ. Then a Σ-interpretation I is extended to a $(\Sigma \cup \{s\})$-interpretation I' by assigning s^I to $\{d \in D_I \mid I \models \psi[d]\}$, where D_I is the domain of I.

Definitions of new function symbols, as in (4.1), require the Σ-interpretation to satisfy totality,

$$\forall x \exists y.\ \phi_f[x, y],$$

and functionality,

$$\forall x, y_1, y_2.\ (\phi_f[x, y_1] \wedge \phi_f[x, y_2]) \Rightarrow y_1 \approx y_2$$

for ϕ_f, in order to be extensible by f. Both of these properties are testable in the initial Σ-interpretation, by virtue of the fact that ϕ_f is a Σ-formula.

When automating the test for consistency of a function definition, the solver does not have access to an interpretation, usually only the axioms AX are given. Testing totality and functionality w. r. t. AX only provides a sufficient condition for consistency, as it is not always the case that every model of AX can be extended to satisfy definitions ϕ_f. In other words, an automated test for consistency could fail although $\mathsf{AX} \cup \phi_f$ is consistent. The test for totality can be circumvented by only taking the $\Leftarrow$-direction of (4.1). This helps with disproving only, as it rules out the trivial case where no models exist of $\mathsf{AX} \cup \mathsf{HYP}$.

The following definition will be used for proving the consistency of definitions extending a specific initial interpretation, without reference to the syntactic form of the definition. This is necessary in order to include recursive definitions, which are not covered by (4.1) and (4.2). The definition aims to capture first-order definitions which just identify existing structures in an interpretation (such as sorted lists or arrays with positive values), rather than adding new values or sorts.

Definition 4.2.1 (I-Admissible Definition)**.** Let Σ be a signature, I a Σ-interpretation, and $f \notin \Sigma$ an operator with an arity over $\mathrm{sort}(\Sigma)$. A set of $(\Sigma \cup \{f\})$-sentences ψ_f is an *I-admissible definition of f* iff I can be expanded to a $(\Sigma \cup \{f\})$-interpretation I' such that the domain of I' is the same as that of I, and $I' \models \psi_f$.

If $\mathcal{M}$ is a class of interpretations (e. g. all first-order models of the set AX), then ψ_f is $\mathcal{M}$-*admissible* if it is I-admissible for all $I \in \mathcal{M}$. If a set ψ_f is admissible w. r. t. all Σ-interpretations, it is Σ-admissible or just 'admissible' for brevity.

Example 4.2.2 (Basic Flat Definitions)**.** Given $f \notin \Sigma$ such that the arity of f is over sorts of Σ, then $\{\forall \overline{x}.\, f(\overline{x}) \approx t[\overline{x}]\}$, where $t[\overline{x}]$ is a Σ-term, is an admissible definition.

Basic flat definitions are useful because they do not need to be checked for totality or functionality– these properties follow from the interpretation function $I : T(\Sigma, \chi) \mapsto D_I$ of the Σ-interpretation I. Basic flat definitions can also be written in the form (4.1): $\forall \overline{x}, y.\, f(\overline{x}) \approx y \Leftrightarrow y \approx t[\overline{x}]$.

Example 4.2.3 (Inadmissible Function)**.** Let $\Sigma = \{c\}$ and take $\mathcal{M}$ to be the class of all Σ-interpretations of cardinality 1. Let $f : D \mapsto D$ and define $\psi_f = \forall x : D.\, f(x) \not\approx f(f(x))$. Then ψ_f is not $\mathcal{M}$-admissible as no $(\Sigma \cup \{f\})$-interpretation satisfies $\forall x : D.\, f(x) \not\approx f(f(x))$ while preserving domains.

Although the definition of admissibility is meant to include recursive definitions, it is not a precise account of 'acceptable' recursive functions. Roughly, a recursive definition is *well-founded* if the recursive application of the definition for any argument value terminates after finitely many steps.

Example 4.2.4. Take $a \in \Sigma$ and $f \notin \Sigma$. Then $\{\forall x.\, f(x) \approx f(f(x))\}$ is admissible although the formula defining f is not well-founded. The function f can always be interpreted as $f(x) \approx a$.

Definitions which depend on other definitions are admissible when the set of definitions can be decomposed into a chain of definitions, each of which depends on a smaller set of definitions down to the base signature. Cyclic definitions are therefore not admissible, although they may be satisfied by some extension of the base signature.

Lemma 4.2.1. *Let* $(\mathsf{Ax}, \mathsf{Def}_{\mathsf{op}_1}, \ldots, \mathsf{Def}_{\mathsf{op}_n})$ *be an extension of* Ax*. Suppose there is a* Σ_0*-model* $I \models \mathsf{Ax}$*. If* $\mathsf{Def}_{\mathsf{op}_i}$ *is an* I*-admissible definition of* op_i *for all* $1 \leq i \leq n$*, then there is a* Σ_n*-interpretation* I' *such that* $I' \models \mathsf{Ax} \cup \bigcup_{1 \leq i \leq n} \mathsf{Def}_{\mathsf{op}_i}$*.*

Proof. By induction over the length n of extensions, using the given model I in the induction start and using admissibility in the induction step. □

Example 4.2.5 (Use of Lemma 4.2.1)**.** Boolean combinations of extensional set definitions (corresponding to intersection, union and complement of sets) are admissible, as are Boolean combinations of admissible predicates, so long as there are no cyclic dependencies among the definitions, i. e. the definitions must be able to be decomposed into a chain of admissible definitions, defined in terms of previous definitions as per Lemma 4.2.1.

As hinted at in the example, Lemma 4.2.1 excludes cyclic dependencies in definitions. This also excludes mutual recursion, e. g. , $\mathsf{even} : \mathbb{Z} \mapsto \mathit{bool}, \mathsf{odd} : \mathbb{Z} \mapsto \mathit{bool}$ defined by:

$$\{\mathsf{even}(0) \land \forall x_{\mathbb{Z}}.\ (\mathsf{even}(1+x) \Leftrightarrow \mathsf{odd}(x)),$$
$$\neg\mathsf{odd}(0) \land \forall x_{\mathbb{Z}}.\ (\mathsf{odd}(1+x) \Leftrightarrow \mathsf{even}(x))\}$$

4.3 Templates for Admissible Recursive Definitions

In the previous section some simple classes of admissible definitions were given. This section describes classes of admissible recursive definitions, which are a necessity when AX axiomatizes recursive data structures, for example. To avoid termination analysis, the admissibility criterion is applied to formulas with specific syntactic form over a fixed theory. Although admissibility is only with respect to models of that theory, such models are common enough to make the syntactic categorization useful.

4.3.1 Admissible Relations

Recursive definitions of relations are admissible when recursive applications of the definition take smaller arguments according to some well-founded order. A consistent interpretation for the defined predicate can be built up by assigning values to the defined predicate in reverse order, from smallest to largest.

Example 4.3.1. The definition $p(x_{\mathbb{Z}}) \Leftrightarrow [(x \geq 1 \Rightarrow p(x-1)) \land (x < 1 \Rightarrow \neg p(x+1))]$ is inadmissible. Simplification of the definition when $x = 0$ yields $p(0) \Leftrightarrow \neg p(1)$, but when $x = 1$ the definition entails $p(1) \Leftrightarrow p(0)$.

Of course some definitions with non-terminating expansion are also satisfiable.

Example 4.3.2. $p(x) \Leftrightarrow \neg p(x-1)$ can be satisfied by either $\{x \in \mathbb{Z} : 2 \mid x\}$ or its complement.

Example 4.3.3. $p(x) \Leftrightarrow \phi \land p(t)$ can always be satisfied by $p^I = \emptyset$ regardless of both t and whether the right-hand side has a terminating expansion.

Definition 4.3.1 (Relativized Definition [Den00])**.** A *relativized definition* w. r. t. a strict well-founded order $<$ on the domain of the Σ-interpretation is a definition of a predicate p:

$$\forall x, \overline{y}.\ p(x, \overline{y}) \Leftrightarrow \phi[x, \overline{y}]$$

such that any $p(z, \overline{s})$ in ϕ appears in the scope of a subformula of the form $\forall z.\ (z < x \Rightarrow \phi')$ or $\exists z.\ (z < x \land \phi')$.

This differs from the definition given in Denecker [Den00] by restricting to the well-founded case only and by assuming a fixed interpretation to be extended.

Example 4.3.4. In the theory of arrays a definition formula in which all index variables (i. e. variables in the second position of read-terms) are guarded by literals that fix a lower bound can be formalized using relativized definitions. For example, to test if an element x is present in the first n indices of an array:

$$\mathsf{contains}(a, x, n_{\mathbb{Z}}) \Leftrightarrow [n > 0 \Rightarrow (\mathsf{read}(a, n) \approx x \vee \exists i_{\mathbb{Z}}.\ i < n \wedge \mathsf{contains}(a, x, i))]$$

Substituting $n - 1$ for i yields a simpler equivalent formula:

$$\mathsf{contains}(a, x, n_{\mathbb{Z}}) \Leftrightarrow [n > 0 \Rightarrow (\mathsf{read}(a, n) \approx x \vee \mathsf{contains}(a, x, n - 1))]$$

Theorem 4.3.1 (Admissibility of Recursive Predicate Definitions)**.** *Let Σ-interpretation I have a well-founded order $<$ on sort S. Let $p \notin \Sigma$ be a predicate symbol whose arity is over* $\mathrm{sort}(\Sigma)$*. Then a relativized definition for p*

$$\overline{\forall}.\ p(x, x_1, \ldots, x_n) \Leftrightarrow \phi$$

where $\mathrm{sort}(x) = S$ *is I-admissible.*

Proof. Assuming a Σ-interpretation I in which D_S is the carrier set for sort S, an interpretation for p can be constructed by induction on D_S. If $d \in D_S$ is minimal w. r. t. $<$, then in $\phi[d]$ all subformulas containing instances of p are guarded by $z < d$, and so they are equivalent to Σ-formulas. Thus $p(d, a_1, \ldots, a_n)$ can be assigned a truth value for any a_i.

Assume for $d' \in D_S$ that for all $d < d'$, $p(d, a_1, \ldots, a_n)$ are assigned truth values for any a_i. Then in $\phi[d']$ every subformula $\forall z.\ (z < x \Rightarrow \phi')$ or $\exists z.\ (z < x \wedge \phi')$ with an instance of p can be evaluated relative to already existing instances of p, by construction. □

Example 4.3.5. Following Example 4.3.1, this definition is not admissible by Theorem 4.3.1, as $x + 1$ appears as an argument of p, yet is not smaller for any valuation of x.

In general, the defined predicate can also appear in the form $p(t[x], a_1, \ldots, a_n)$, where $t[x]$ is a term such that $t < x$ in the appropriate order. For example, t could be $x - 1$ in $T_{\mathbb{Z}}$. To justify this, suppose such a term t is in $\phi[p(t[x])]$, such that $t[x] < x$ is true in the Σ-interpretation I. By abstraction, $\forall z.\ (z = t[x] \Rightarrow \phi[p(z)])$ is equivalent to the original formula. Similarly, $\forall z.\ (z = t[x] \Rightarrow z < x)$ is obtained by abstraction of $t[x] < x$. Then $\forall z.\ (z < x \Rightarrow (z = t[x] \Rightarrow \phi[p(z)]))$ is equivalent to $\phi[p(t[x])]$ and it has the required form for a relativized definition.

Lemma 4.3.1. *Given an interpretation I with well-founded order $<$, as in Definition 4.3.1, a definition of a predicate p:*

$$\forall x, \overline{y}.\ p(x, \overline{y}) \Leftrightarrow \phi[x, \overline{y}]$$

such that, for any instance $p(t, \overline{s})$ in ϕ and any valuation ν, $I \models \nu(t) < \nu(x)$ is relativized.

The following example applies this lemma to T_{LIST}.

Definition 4.3.2 (Sublist Order)**.** Define the order $<_{\mathsf{LIST}}$ on list constructor terms $\mathsf{cons}(s,t)$ as the transitive closure of $\{(l_1,l_2) \mid \exists x.\ l_2 = \mathsf{cons}(x,l_1)\}$. For all *acyclic* models of the list axioms this order is well-founded.

Example 4.3.6 (Relativized List Predicates)**.** Let Def_p be a formula of the form

$$\forall l_{\mathsf{LIST}}, \bar{s}.\ p(l,\bar{s}) \Leftrightarrow$$
$$l \approx \mathsf{nil} \wedge B[\bar{s}] \tag{P1}$$
$$\vee\ \exists h_{\mathbb{Z}},\ t_{\mathsf{LIST}}.\ l \approx \mathsf{cons}(h,t) \wedge C[\bar{s},h,t] \tag{P2}$$

where B is a $\Sigma \cup \{p\}$-formula not containing p, and C is a $\Sigma \cup \{p\}$-formula possibly containing $p(t,\overline{s'})$, but not $p(\mathsf{cons}(h,t),\overline{s'})$.
Since $l \approx \mathsf{cons}(h,t)$ implies $t <_{\mathsf{LIST}} l$, the relativized form of (P2):

$$\exists t_{\mathsf{LIST}}.\ t <_{\mathsf{LIST}} l \wedge \exists h_{\mathbb{Z}}.\ l \approx \mathsf{cons}(h,t) \wedge C[k,h,t]$$

is equivalent to (P2) as above. Therefore Def_P formulas are relativized formulas.

This can also be extended to theories of acyclic recursive data structures.

As for lists, the sub-structure relation $<_{\mathsf{RDS}}$ defined as the transitive closure of $\{(r_1,r_2) : r_2 = c(\ldots,r_1,\ldots)\}$, is well-founded for acyclic models of $\mathsf{AX}_{\mathsf{RDS}}$. Therefore, predicates defined similarly to Def_p for Σ_{RDS} are admissible.

In Armando et al. [ABRS09] it is observed that Superposition calculi with an appropriate term order can finitely saturate the theory of records (i. e. T_{RDS} without recursion in the arguments to the constructor function).

4.3.2 Admissible Functions

For functions, not only must the recursion be well-founded, but the defining formula must be functional and total also. Now that the defining formula possibly contains the symbol being defined, these properties cannot simply be evaluated in the interpretation that is being extended. This section applies the criteria for a relativized definition to the case of function definitions and also gives specialized descriptions for the case of lists.

Replacing the predicate definition on the left side of the relativized definition formula gives a condition for admissible function definitions w. r. t. a well-founded order:

Definition 4.3.3 (Relativized Function Definition)**.**

$$f(x) \approx y \Leftarrow \phi[x,y]$$

where occurrences of $f(s)$ in subformula G of ϕ are of the form $\forall z.\ z < x \leftarrow G[z]$ or $\exists z.\ z < x \wedge G[z]$, and $<$ is interpreted as a well-founded order.

The check for totality is avoided by using ϕ as a sufficient condition only, i. e. an implication not an equivalence. In practice, this means that if a formula does not

completely specify the new symbol, then it is completed arbitrarily for the unspecified values. Functionality must still be shown i. e.

$$\forall x, y_1, y_2.\ \phi[x, y_1] \wedge \phi[x, y_2] \Rightarrow y_1 \approx y_2 \tag{4.3}$$

must hold in the extended interpretation, if it is to satisfy the relativized definition. The test for (4.3) can be partially automated by testing whether a stronger property is satisfied by all models of $\mathsf{AX} \cup \mathsf{HYP}$. The stronger property is arrived at by replacing all subterms of the form $f(s)$ in (4.3) with fresh variables. This new formula does not contain the symbol f and can be checked by a theorem prover for validity. If it is valid, then it holds for all models and all values of the substituted variables, specifically (4.3) is satisfied in the extended interpretation. For some definitions it might not be the case that the stronger property is satisfied in all models.

Theorem 4.3.2 (Admissibility of Recursive Function Definitions)**.** *Given Σ-interpretation I with a with a well-founded order $<$ on sort S, let $f \notin \Sigma$ be a function symbol whose arity is over* $\mathrm{sort}(\Sigma)$*. Given a definition*

$$f(x_1, \ldots, x_i, \ldots, x_n) \approx y \Leftarrow \phi$$

where $\mathrm{sort}(x_i) = S$ *and ϕ is a formula in which any occurrence of a term $f(r_1, \ldots, r_i, \ldots, r_n)$ is such that $\nu_x(r_S) < \nu_x(x_i)$ for any valuation ν_x of x_i. Let ϕ' be ϕ in which every subterm $f(\bar{r})$ is replaced by a fresh variable. If $I \models \overline{\forall}(\phi'[y/y_1] \wedge \phi'[y/y_2]) \Rightarrow y_1 \approx y_2$, then the definition of f is I-admissible.*

Example 4.3.7. Integer multiplication can be defined in terms of addition by using recursion:

$$\begin{aligned} x * y \approx z \Leftarrow\ & \\ & (y \approx 0 \Rightarrow z \approx 0) \wedge \\ & (y > 0 \Rightarrow z \approx x * (y - 1) + x) \wedge \\ & (y < 0 \Rightarrow z \approx x * -y) \end{aligned}$$

A definition of integer multiplication is clearly $\mathcal{P}$-admissible, where $\mathcal{P}$ is the standard interpretation of Presburger arithmetic.

This is useful because first-order solvers, such as *Beagle*, typically have theory reasoners only for Presburger arithmetic, as that theory is decidable. Multiplication is modelled by adding axioms from the example above to input formula sets that require multiplication. Quite often counter-satisfiable conjectures using multiplication are easily proven contradictory, while their negation is not: for example, the conjecture $\forall x, y.\ x * x \approx y$ (easily shown unsatisfiable) becomes $\exists x, y.\ x * x \not\approx y$ when negated. The latter form is satisfiable, and so termination of proof search is unlikely.

Theorem 4.3.2 provides the general case description for admissible recursive functions. It is applied to specific sets of interpretations by using properties of the well-

founded order found in the target interpretations. For recursive data structures the sequence of constructor applications gives the order, and so order predicates can be replaced appropriately. This is illustrated in the following examples.

Example 4.3.8. Given an acyclic model I_A of $\mathsf{AX}_{\mathsf{LIST}}$, it is possible to extend with the definition N_{length}:

$$\mathsf{length}(l_{\mathsf{LIST}}) \approx y_{\mathbb{Z}} \Leftarrow [(l \approx \mathsf{nil} \to y \approx 0) \wedge (l \not\approx \mathsf{nil} \to y \approx 1 + \mathsf{length}(\mathsf{tail}(l)))]$$

For lists l in a Σ_{LIST}-interpretation $\mathsf{tail}(l) <_{\mathsf{LIST}} l$, and for I_A this is well-founded. Functionality follows from the extensionality axiom for cons. Therefore, N_{length} is I_A-admissible. In general, N_{length} is not Σ_{LIST}-admissible, as it is never admissible in a cyclic model of lists.

Example 4.3.9. Consider another definition N_{rep}:

$$\mathsf{rep}(x) \approx y \Leftarrow \mathsf{cons}(x, \mathsf{rep}(x)) \approx y$$

This is not a relativized definition, since the argument to the recursive term $\mathsf{rep}(x)$ does not decrease relative to $<_{\mathsf{LIST}}$. Given an acyclic model I_A of $\mathsf{AX}_{\mathsf{LIST}}$, then N_{rep} is not I_A-admissible, however it is I_C-admissible where I_C is a model of $\mathsf{AX}_{\mathsf{LIST}}$ with cyclic lists.

The general form Theorem 4.3.2 can be specialized to T_{LIST} by replacing the general well-founded order with $<_{\mathsf{LIST}}$. Typical applications of the list theory usually assume acyclicity, so we introduce a schema for admissible LIST functions which excludes definitions of cyclic lists such as N_{rep} above, which would prevent using $T_{\mathbb{Z}}$ as an element theory.

Let $\Sigma \supseteq \Sigma_{\mathsf{LIST}}$ be a signature, $S \in \mathrm{sort}(\Sigma)$ and $f \notin \Sigma$ a function symbol with arity $\mathbb{Z} \times \mathsf{LIST} \mapsto S$. Let Def_f be a set of (implicitly) universally quantified formulas of the form below, where $\bar{k}$ is a tuple of non-list variables h is $\mathbb{Z}$-sorted and t is LIST-sorted:

$$f(\bar{k}, \mathsf{nil}) \approx b[\bar{k}] \Leftarrow B[\bar{k}] \tag{f_0}$$

$$f(\bar{k}, \mathsf{cons}(h,t)) \approx c_1[\bar{k}, h, t, f(\bar{k}, t)] \Leftarrow C_1[\bar{k}, h, t, f(\bar{k}, t)] \tag{f_1}$$

$$\vdots$$

$$f(\bar{k}, \mathsf{cons}(h,t)) \approx c_n[\bar{k}, h, t, f(\bar{k}, t)] \Leftarrow C_n[\bar{k}, h, t, f(\bar{k}, t)] \tag{f_n}$$

where B is a Σ-formula. All of C_i and c_i are $(\Sigma \cup \{f\})$-formulas and terms respectively, as they all contain f. Each definition must contain a base case (f_0) in order to be well-founded.

Lemma 4.3.2. *Let I_A be a Σ^+-interpretation that satisfies the acyclic property on Σ_{LIST}. If for all $1 \leq i < j \leq n$ the formula*

$$\forall k_{\mathbb{Z}}\ h_{\mathbb{Z}}\ t_{\mathsf{LIST}}\ x_s.\ (C_i[k,h,t,x] \wedge C_j[k,h,t,x]) \Rightarrow c_i[k,h,t,x] \approx c_j[k,h,t,x]$$

is LIST*-valid then* Def_f *is an I_A-admissible definition of f w.r.t. Σ^+.*

Proof. To show I_A-admissibility, extend I_A with a new function f^I that interprets f and satisfies Def_f. By virtue of the axiom of construction

$$x \approx \mathsf{nil} \vee x \approx \mathsf{cons}(h(x), t(x)),$$

structural induction over nil and cons-terms is adequate. Base case: given $\bar{s}$, both $b[\bar{s}]$ and $B[\bar{s}]$ are over Σ^+, and both are assigned values by I_A:

$$f^I(\bar{s}, I_A(\mathsf{nil})) = \begin{cases} I_A(b[\bar{s}]), & \text{if } I_A(B[\bar{s}]) \text{ is true,} \\ e_0, & \text{otherwise} \end{cases}$$

If $I_A(B[\bar{s}])$ is false, then an arbitrary value e_0 can be assigned.

Next, assume for some cons-term l that $f^I(\bar{s}, l')$ is defined for all $\bar{s}$ and l' less than l in the sub-list order. In each of the conditions C_i for formulas (f_i), the f terms are over smaller LIST terms, so they are also evaluable at this stage in the induction. If no C_I is true, $f^I(\bar{t}, l)$ is assigned a default value; if C_i is true, then $f^I(\bar{t}, l) = c_i[\bar{t}]$. Otherwise C_i and C_j are true, but by the condition in the lemma, this means that $c_i = c_j$ and f^I can be safely assigned that value. Finally, since I_A is acyclic, the induction covers all LIST-terms and f^I is therefore total. □

4.3.3 Higher Order LIST Operations

This section demonstrates the usefulness of Lemma 4.3.2 by analyzing some higher order functions of lists (similar arguments apply for recursive data structures), and shows that first-order translations of applications of the given morphisms are I_A-admissible. Perhaps the most simple of these is map_f, which applies a function f to each element of the list. In order to support a wider range of operations (for example those producing nested lists), and to simplify presentation, we will work with an unsorted logic just for this section. In each of the following, operators are parameterized by an admissible function f, a higher-order argument in usual programming practice.

$$\mathsf{map}_f(\mathsf{nil}) \approx \mathsf{nil} \wedge$$
$$\mathsf{map}_f(\mathsf{cons}(x, ys)) \approx \mathsf{cons}(f(x), \mathsf{map}_f(ys))$$

As map_f is condition free, and so long as f is admissible, then map_f is I_A-admissible too.

The function append is used as a helper function in flatMap:

$$\mathsf{append}(\mathsf{nil}, l_2) \approx \mathsf{nil} \wedge$$
$$\mathsf{append}(\mathsf{cons}(x, ys), l_2) \approx \mathsf{cons}(x, \mathsf{append}(ys, l_2))$$

Since each condition is empty, this is immediately I_A-admissible.

$$\mathsf{flatMap}_f(\mathsf{nil}) \approx \mathsf{nil} \wedge$$
$$\mathsf{flatMap}_f(\mathsf{cons}(x, ys)) \approx \mathsf{append}(f(x), \mathsf{flatMap}_f(ys))$$

$$\mathsf{fold}_f(\mathsf{nil}, b) \approx b \wedge$$
$$\mathsf{fold}_f(\mathsf{cons}(x, ys), b) \approx f(x, \mathsf{fold}_f(ys, b))$$

Again, this does not require any further work to be I_A-admissible.

4.4 Applications

In general, it will be difficult to automatically discover admissibility of formulas 'in the wild': first one must settle on a theory or satisfiable axiom set, then select the correct chaining of definition sets and appropriate structure.... Rather, the methods and results given here provide a library of already admissible definitions (or templates for proving admissibility) over common first-order theories that users of refutation complete solvers can use in their own theorem proving applications. Additionally, these methods could be used to integrate refutation complete first-order theorem provers in larger systems, (proof assistants for higher-order logic typically), in a way that extends their present capabilities. In these applications knowledge about the admissibility of sets of axioms or definitions may already exist; then, testing both CON and $\neg$CON in parallel can allow one to conclude that CON is a theorem or non-theorem depending upon which proof terminates first; obviously this is not the case if CON is contingently true.

4.4.1 Non-theorems in T_{LIST}

Baumgartner and Bax [BB13] give a selection of admissible definitions (although using a slightly different definition of admissibility, they remain admissible with the new definition) which were tested with a selection of first-order reasoners which have built-in arithmetic reasoning capability. Those results are updated here with a larger selection of reasoners.

The following definitions extend Σ_{LIST} with new functions and predicates. They can be shown to be admissible using the lemmas and theorems of the previous section. The function length is as defined in Example 4.3.8.

Together they will be used to disprove conjectures in the extended list theory with integer elements. The goal is to demonstrate that conjectures on which reasoners do not usually terminate (due to satisfiability in an infinite cardinality theory) can be disproved using the methods described here.

Let $\mathsf{count} : \mathbb{Z} \times \mathsf{LIST} \mapsto \mathbb{Z}$, $\mathsf{append} : \mathsf{LIST} \times \mathsf{LIST} \mapsto \mathsf{LIST}$ and $\mathsf{in} : \mathbb{Z} \times \mathsf{LIST}$ be op-

erators. Consider the extension of $\mathsf{AX}_{\mathsf{LIST}}$ with the following (admissible) definitions.

$$\begin{aligned}
\mathsf{count}(k, \mathsf{nil}) &\approx 0 & \mathsf{append}(\mathsf{nil}, l) &\approx l \\
\mathsf{count}(k, \mathsf{cons}(h, t)) &\approx \mathsf{count}(k, t) \Leftarrow k \not\approx h & \mathsf{append}(\mathsf{cons}(h, t), l) &\approx \mathsf{cons}(h, \mathsf{append}(t, l)) \\
\mathsf{count}(k, \mathsf{cons}(h, t)) &\approx \mathsf{count}(k, t) + 1 \Leftarrow k \approx h
\end{aligned}$$

The function count counts the occurrences of integer k in the given list, while append creates a new list by appending the second argument list to the end of the first list. Of the list functions, only count requires an application of Lemma 4.3.2 as both append and length have no side conditions. The proof of functionality of count is straightforward, as $k \approx h$ and $k \not\approx h$ cannot be true simultaneously, so it is admissible by the lemma.

$$\begin{aligned}
\mathsf{inRange}(n, l) \Leftrightarrow\ & l \approx \mathsf{nil} \vee \\
& (0 \leq \mathsf{head}(l) \wedge \mathsf{head}(l) < n \wedge \mathsf{inRange}(n, \mathsf{tail}(l))) \\
\mathsf{in}(k, l) \Leftrightarrow\ & \mathsf{count}(k, l) > 0
\end{aligned}$$

The conjectures given in the following table are false in the (acyclic) theory of lists. Note that all free variables are assumed to be universally quantified. Since the definitions of the functions inRange, length, etc. are admissible, solvers can deduce satisfiability of the statements shown by deriving a contradiction. In the notation used in the introduction, AXIOM $\cup$ HYP $\cup$ CON is unsatisfiable, while AXIOM $\cup$ HYP $\cup \neg$CON is satisfiable. Results of running the provers on the former appear in the "Sat" (for satisfiability) column, while the latter appear in the "Ref" column (for refutation).

Solvers used were *Beagle* (0.9.51) and Z3 (4.5.1) both with default settings (*Beagle* in automatic mode) and with a time limit of 60 seconds. Columns are marked only when the solver returns the correct result in the time limit.

	Superpos.		SMT	
Problem	Ref	Sat	Ref	Sat
$\mathsf{inRange}(4, \mathsf{cons}(1, \mathsf{cons}(5, \mathsf{cons}(2, \mathsf{nil}))))$		x		x
$n > 4 \Rightarrow \mathsf{inRange}(n, \mathsf{cons}(1, \mathsf{cons}(5, \mathsf{cons}(2, \mathsf{nil}))))$		x		x
$\mathsf{inRange}(n, \mathsf{tail}(l)) \Rightarrow \mathsf{inRange}(n, l)$		x		
$\exists n, l.\ l \not\approx \mathsf{nil} \wedge \mathsf{inRange}(n, l) \wedge n - \mathsf{head}(l) < 1$		x		x
$\mathsf{inRange}(n, l) \Rightarrow \mathsf{inRange}(n - 1, l)$		x		
$l \not\approx \mathsf{nil} \wedge \mathsf{inRange}(n, l) \Rightarrow n - \mathsf{head}(l) > 2$		x		x
$0 < n \wedge \mathsf{inRange}(n, l) \wedge l' \approx \mathsf{cons}(n - 2, l) \Rightarrow \mathsf{inRange}(n, l')$		x		x
$\mathsf{length}(l_1) \approx \mathsf{length}(l_2) \Rightarrow l_1 \approx l_2$		x		x
$n \geq 3 \wedge \mathsf{length}(l) \geq 4 \Rightarrow \mathsf{inRange}(n, l)$		x		
$\mathsf{count}(n, l) \approx \mathsf{count}(n, \mathsf{cons}(1, l))$		x		
$\mathsf{count}(n, l) \geq \mathsf{length}(l)$		x		x
$l_1 \not\approx l_2 \Rightarrow \mathsf{count}(n, l_1) \not\approx \mathsf{count}(n, l_2)$		x		x
$\mathsf{length}(\mathsf{append}(l_1, l_2)) \approx \mathsf{length}(l_1)$		x		x
$\mathsf{length}(l_1) > 1 \wedge \mathsf{length}(l_2) > 1 \Rightarrow \mathsf{length}(\mathsf{append}(k, l)) > 4$		x		x
$\mathsf{in}(n_1, l_1) \wedge \neg\mathsf{in}(n_2, l_2) \wedge l_3 \approx \mathsf{append}(l_1, \mathsf{cons}(n_2, l_2)) \Rightarrow \mathsf{count}(n, l_3) \approx \mathsf{count}(n, l_1)$				

4.4.2 Non-theorems in T_{ARRAY}

For arrays, the defined predicates are: $\mathsf{distinct} : \mathsf{ARRAY} \times \mathbb{Z} \mapsto \mathit{Bool}$ is true if the first n elements are unique; $\mathsf{sorted} : \mathsf{ARRAY} \times \mathbb{Z} \mapsto \mathit{Bool}$, where $\mathsf{sorted}(a, n)$ is true if the first n elements of a are sorted in increasing order; $\mathsf{inRange} : \mathsf{ARRAY} \times \mathbb{Z} \times \mathbb{Z}$, as for lists, $\mathsf{inRange}(a, r, n)$ is true if the first n elements fall in the range $[0, r]$. None of the definition formulas are recursive, and so they fit the description of an admissible predicate. This does not require further proof before it can be used. In order to fit with typical use cases, the predicates are restricted to array prefixes. It seems nonsensical to require that an array is only sorted if it is sorted across infinitely many indices, other properties also only make sense when restricted to a prefix.

$$\begin{aligned}
&\mathsf{inRange}(a, r, n) \Leftrightarrow \\
&\quad \forall i.\ (0 \leq i \wedge i < n) \\
&\quad \Rightarrow (r \geq \mathsf{read}(a, i) \wedge \mathsf{read}(a, i) \geq 0)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{distinct}(a, n) \Leftrightarrow \\
&\quad \forall i, j.\ (n > i \wedge n > j \wedge j \geq 0 \wedge i \geq 0) \\
&\quad \Rightarrow \mathsf{read}(a, i) \approx \mathsf{read}(a, j) \Rightarrow i \approx j)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sorted}(a, n) \Leftrightarrow \\
&\quad \forall i, j.\ (0 \leq i \wedge i < j \wedge j < n) \\
&\quad \Rightarrow \mathsf{read}(a, i) \leq \mathsf{read}(a, j)
\end{aligned}$$

Definitions of array functions have the following arities and uses:
$\mathsf{rev} : \mathsf{ARRAY} \times \mathbb{Z} \mapsto \mathsf{ARRAY}$ returns a copy of an array with the order of the first n elements reversed; $\mathsf{max} : \mathsf{ARRAY} \mapsto \mathbb{Z}$, returns the maximal element in the first n entries. Again, they are not recursive and so a proof of functionality is required. Note that in order to ensure functionality, the behaviour of rev must be specified outside of the given prefix as well, as it returns an array. Both solvers were able to prove functionality of both definitions.

$$\begin{aligned}
&\mathsf{rev}(a, n) \approx b \Leftarrow \\
&\quad \forall i.\ ((0 \leq i \wedge i < n) \Rightarrow \mathsf{read}(b, i) \approx \mathsf{read}(a, n - (i + 1))) \\
&\quad \vee\ ((0 > i \vee i \geq n) \wedge \mathsf{read}(b, i) \approx \mathsf{read}(a, i)) \\
&\mathsf{max}(a, n) \approx w \Leftarrow \\
&\quad \forall i.\ ((0 \leq i \wedge i < n) \Rightarrow w \geq \mathsf{read}(a, i)) \\
&\quad \wedge\ \exists j.\ (n > j \wedge j \geq 0 \wedge \mathsf{read}(a, j) \approx w)
\end{aligned}$$

The conjectures given in the following table are false in the extensional theory of arrays. Again, the definitions are admissible, solvers can deduce satisfiability of the statements shown by deriving a contradiction. In the notation used in the introduction, $\mathsf{AXIOM} \cup \mathsf{HYP} \cup \mathsf{CON}$ is unsatisfiable, while $\mathsf{AXIOM} \cup \mathsf{HYP} \cup \neg\mathsf{CON}$ is satisfiable. Results of running the provers on the former appear in the "Sat" (for satisfiability) column, while the latter appear in the "Ref" column (for refutation).

Solvers used were *Beagle* (0.9.51) and Z3 (4.5.1), both with default settings (*Beagle* in automatic mode) and with a time limit of 60 seconds. Columns are marked only when the solver returns the correct result in the time limit.

	Superpos.		SMT	
Problem	Ref	Sat	Ref	Sat
$n \geq 0 \Rightarrow \mathsf{inRange}(a, \mathsf{max}(a, n), n)$		x	*	
$\mathsf{distinct}(\mathsf{init}(n), i)$		x	*	
$\mathsf{read}(\mathsf{rev}(a, n+1), 0) \approx \mathsf{read}(a, n))$		x	*	
$\mathsf{distinct}(a, n) \Rightarrow \mathsf{distinct}(\mathsf{rev}(a, n))$		x	*	*
$\exists n_{\mathbb{Z}}.\ \neg\mathsf{sorted}(\mathsf{rev}(\mathsf{init}(n), m), m)$		x	*	*
$\mathsf{sorted}(a, n) \wedge n > 0 \Rightarrow \mathsf{distinct}(a, n)$		x	*	

The SMT results columns are marked specially since Z3 could not solve any of the problems when using the problem statements given above. In particular, the incomplete specification of rev and max caused Z3 to report "unknown" for all problems. This is correct where the answer is "satisfiable"; in the absence of the meta-level arguments for satisfiability of admissible functions one cannot be sure that a function exists satisfying the given property. This is related to the sufficient completeness problem described in the previous chapter.

However, given an explicit definition for the two functions, e. g.

$$\begin{aligned}
&\mathsf{maxInner}(a, n, c) := \\
&\quad \mathsf{maxInner}(a, n-1, \mathsf{max}_{\mathbb{Z}}(c, \mathsf{read}(a, n))) \\
&\mathsf{max}(a, n) := \\
&\quad \mathsf{maxInner}(a, n, \mathsf{read}(a, n))
\end{aligned}$$

for max, Z3 was able to correctly solve the problems marked with (*) in the table. Together these provide a nice illustration of the respective complementary strengths of the two solvers.

4.4.3 TPTP Arithmetic non-theorems

As shown in Example 4.3.7 multiplication is admissible. As a result, problems in which linear arithmetic theories, $T_{\mathbb{Z}}$ or $T_{\mathbb{Q}}$, are extended with multiplication can be dealt with using the method suggested here, i. e. proving the conjecture is a non-theorem. Using this simple method, *Beagle* won the typed non-theorem division of CASC-J8, solving 10/50 satisfiable problems in the TPTP. However, this only improved on the second-best score by 1 solved problem. This is by no means a complete answer to the problem of first-order satisfiability, notably, it never produces a solution (as in, a counter-model).

Table 4.4.3 has a comparison of solving times for the regular (Ref) and non-negated (Sat) form of the TPTP problems solved by *Beagle* . All problems are (counter) satisfiable in their default form with conjectures negated, this is the Ref column; without negation all result in an unsatisfiable clause set, implying the conjecture is a non-theorem. Both runs had a 60 second timeout; any empty entries did not terminate in the time limit.

Problems `ARI536=3` and `ARI575=3` rely on Z3 to discharge pure clauses in the theory $T_{\mathbb{R}}$, all others use the built in LIA solver. The 'ns' entry denotes a run where

Problem	Ref	Sat
ARI126=1	-	2.3
ARI127=1	-	3.6
ARI536=3	ns	1.5
ARI575=2	0.1	0.9
ARI575=3	0.1	1.0
NUM879=1	-	1.2
NUM880=1	-	1.3
NUM881=1	0.7	1.3
NUM885=1	-	1.2
NUM886=1	-	1.1

Table 4.1: Solving time (s) when conjecture is negated (Ref) and not negated (Sat).

Beagle produced a saturation but could not conclude 'Satisfiable' due to the presence of uninterpreted BG sorted symbols.

4.4.4 Definitions in SMT-Lib format

The SMT-lib 2.5 standard [BFT15] provides syntax specifically for giving definitions of new symbols, possibly using recursion, and the specific form of the definition guarantees both functionality and totality. Applications of prover technology emit goals in SMT-lib syntax (e. g. Isabelle, Why3), and first-order solvers support SMT-lib syntax either by translation or directly.

The SMT-lib expression `(define-fun f` $((x_1 S_1) \ldots (x_n S_n))$ `S t)` defines a function with arity $f : S_1 \times \ldots \times S_n \mapsto S$; assuming that f does not appear in term t and $\mathrm{sort}(t) = S$. It is equivalent to the formula $\forall x_1, \ldots, x_n.\ f(x_1, \ldots, x_n) \approx t$. The command `define-funs-rec` allows multiple function definitions in a single statement (to have mutual recursion) and it allows recursive usage of defined symbols.

Note that the definition is accomplished by equating the defined term to a single term t. Definitions involving side conditions of the form in Definition 4.3.3 or its specialization to lists, are modelled by if-then-else terms: $ite(\phi, r, s)$, where ϕ is a formula (the condition), and r, s are terms of the same sort. If-then-else terms can be translated into FOL:

$$F[ite(\phi, r, s)] \text{ becomes } (\phi \Rightarrow F[r]) \wedge (\neg\phi \Rightarrow F[s])$$

for some formula F. Conditions in if-then-else terms translate to perfect dichotomies, i. e. ϕ guards the 'if' term and $\neg\phi$ guards the 'else' term, so `define-fun` definitions are well-formed and total by default. For example $\mathsf{signum} : \mathbb{Z} \mapsto \mathbb{Z}$ defined by

if-then-else terms and FOL:

$$\begin{aligned} \mathsf{signum}(x) &\approx \mathit{ite}(x > 0, 1, \mathit{ite}(x \approx 0, 0, -1)) \\ \mathsf{signum}(x) &\approx 1 \Leftarrow (x > 0) \\ \mathsf{signum}(x) &\approx 0 \Leftarrow (x \leq 0 \wedge x \approx 0) \\ \mathsf{signum}(x) &\approx -1 \Leftarrow (x \leq 0 \wedge x \not\approx 0) \end{aligned}$$

Of course, definitions by `define-funs-rec` must first be shown to be well-founded.

In many theorem-proving applications it is usually not the case that problems consist of just axioms, definitions, and conjecture formulas. Other formulas may specify extra properties or lemmas about the problem at hand. In other words, the problem might have structure: $\mathsf{AX} \cup \mathsf{DEF} \cup \mathsf{HYP} \models \mathsf{CON}$, where HYP is a set of arbitrary formulas that are neither axiomatic nor fit the syntactic criteria for definitions. The method above can be applied by moving the formulas HYP to the right of the consequence relation: $\mathsf{AX} \cup \mathsf{DEF} \models (\bigwedge \mathsf{HYP}) \Rightarrow \mathsf{CON}$, then the fact that $\mathsf{AX} \cup \mathsf{DEF}$ is consistent can be used. Specifically, if a refutation based theorem prover can derive a contradiction from

$$\mathsf{AX} \cup \mathsf{DEF} \cup (\mathsf{HYP} \Rightarrow \mathsf{CON}) \tag{4.4}$$

then one can conclude $\mathsf{AX} \cup \mathsf{DEF} \models \bigwedge \mathsf{HYP} \wedge \neg \mathsf{CON}$.

The disadvantage is that the negated form of HYP in (4.4) may not be in a well-behaved fragment such as the array property or quantifier-free fragment. This would affect methods that attempt to prove satisfiability, but refutation based methods are affected by a different set of properties.

4.5 Summary

This chapter presents a syntactic criterion for definitions which preserve satisfiability of axiom sets. This is specialized for recursive definitions, assuming a reference model with some fixed well-founded order. Then, standard theories can be extended by new definitions which can be checked automatically, or manually and then re-used. One class of automatically recognizable definitions is given by SMT-lib style `define-fun` specifications.

The method is used to show counter-satisfiability of non-theorems over standard theories extended with new definitions, using both an SMT solver and a Superposition solver. For problems over lists, the counter-satisfiability method was able to show that the hypothesis was a non-theorem, while the usual refutation method did not terminate.

For problems over arrays, the counter-satisfiability method prevailed for the Superposition solver, when reasoning with an implicit description of a function. SMT performed better when using an explicit description of the new function, and could disprove the conjecture in the usual refutation setting (i. e. where the conjecture is negated).

Moreover, the counter-satisfiability method provides an alternative for reasoning with multiplication over integers and rationals, even though only the additive theory of each is decidable. By assuming the built-in definition of multiplication is satisfiable, any contradiction produced by a conclusion implies that its negation is a theorem, and so the conjecture is counter-satisfiable.

4.5.1 Related Work

Many common formulas are not included in the array property fragment (Chapter 2, and also [BMS06]), for example an injectivity predicate for arrays, see distinct in the previous section. Ghilardi et al. [GNRZ07] provide a decision procedure for an extension of the array theory and demonstrate how decision procedures may be derived for extensions to this theory, many of which lie outside the array property fragment. This relies on the existence of a 'standard model' for the theory and extension, whose existence must be demonstrated a priori.

In contrast to these works, we do not provide decision procedures for specific fragments. This is intentionally so, in order to support disproving tasks in the presence of liberally formulated additional axioms (the set HYP above). Although we employ Superposition based provers in the experiments, like some approaches above, our approach does not hinge on finite saturation. Claessen and Lilliestr\"om [CL11] present a method for showing that a set of formulas does not admit finite models. It does not answer whether infinite models exist, and so is complementary to the above. Suter et al. [SKK11b] give a semi-decision procedure for checking satisfiability of correctness properties of recursive functional programs on algebraic data types, which overlaps with the given method on lists (Lemma 4.3.2) by imposing similar syntactic restrictions. Their method works differently, by partial unrolling of function definitions into quantifier-free logic, instead of theorem proving on (quantified) formulas.

Ge and de Moura [GdM09] describe *macro definitions*. A macro is a non-ground clause $g(\overline{x}) \approx t[\overline{x}]$ where g does not occur in t. They suggest that the best way to deal with terms $g(\overline{s})$ is to remove them entirely from the input formula, after which the clause defining g is equivalent to true. They generalize this to the concept of a *pseudo-macro* which is a symbol g defined by a set of clauses $D_g = \{C_1[\overline{x}], \ldots, C_n[\overline{x}]\}$ such that all C_i contain $g(\overline{x})$ and are trivially true after replacing $g(\overline{x})$ with some term $t_g[\overline{x}]$. Another simple form of pseudo macro is $D_g = \{C_1[\overline{x}] \vee g(\overline{x}) \bowtie t_g[\overline{x}], \ldots, C_n[\overline{x}] \vee g(\overline{x}) \bowtie t_g[\overline{x}]\}$ where $\bowtie$ is $\approx, \leq$ or $\geq$. This concept is exploited to limit instantiation in the SMT scheme they describe. Note that macros fit the pattern of basic definitions described in Example 4.2.2, and so pseudo-macros could offer a generalization along the same lines.

Reynolds et al. [RBCT16] give an admissibility criterion for use in translating recursive function definitions for consumption by SMT solvers. This criterion identifies when the translation in question preserves unsatisfiability of the function definition. Although similar in intent, this definition of admissibility is semantic and requires an external proof of admissibility. Well-founded definitions are shown to be admissible, so only a termination proof is required for those definitions.

In particular, a definition is admissible in the sense of Reynolds et al. when expansion with the terms of the definition does not affect T-satisfiability of a set of formulas that uses the definitions. It is likely that this is a more general account of admissibility than that given here, for example, definitions identified in Theorem 4.3.1 are (semantically) admissible, by virtue of being well-founded. Nevertheless, syntactic criteria are useful in that they give a short-cut method of proving the admissibility of the definition, although they may not cover all possible expressions of that property.

Chapter 5

Finite Quantification in Hierarchic Theorem Proving

5.1 Motivation

The previous chapter addressed the problem of disproving contradictory conjectures in the presence of background theories. This chapter considers the obvious next question: what to do when the conjecture is contingently true, in other words, when $\mathsf{HYP} \cup \{\neg\mathsf{Con}\}$ is $\mathcal{B}$-satisfiable. In particular, under the assumption that there are only finitely many free BG-sorted subterms in the ground instances of the clause set (more specifically, the *relevant* terms are finite), then the hierarchic satisfiability problem can be solved using Superposition for hierarchic theories as described in Chapter 2.

This chapter also describes an algorithm for the hierarchic satisfiability problem that employs a *conflict-guided* instantiation strategy for producing formulas that are free of the completeness problems that can lead to an incorrect conclusion of satisfiability. Unlike traditional finite model finders, it avoids exhaustive instantiation, hence it is expected to scale better with the size of the problem domains. While aimed at demonstrating satisfiability, if the algorithm determines unsatisfiability w. r. t. finite domains, the given clause set is also unsatisfiable w. r. t. unbounded domains. Then this approach could be seen as an extension of quantifier instantiation heuristics that determines satisfiability w. r. t. finite domains.

The key results of the chapter are a correctness proof and experimental results that illustrate the performance characteristics of the algorithm. This updates results in Baumgartner et al. [BBW14] and places them in context of later developments.

Section 5.2 contains a step-by-step application of the satisfiability procedure to an example problem in the theory of arrays. Then the particular language fragment used to model the Ground Base-sorted Term (GBT)-fragment is introduced, as well as a (previously unpublished) technique for modelling quantification over arbitrary finite sets using finite integer sets. The satisfiability procedure is introduced in Section 5.4, as well as a heuristic that uses solvers to find terms for updating the equivalence relation. Section 5.5 contains a small set of experiments that illustrate the range of possible behaviours and the scalability of the algorithm. Finally, Section 5.6 places the

satisfiability procedure in the context of a selection of other satisfiability procedures that include theory reasoning.

5.1.1 Overview

While the first-order validity problem is semi-decidable, the satisfiability problem is not, as there is no way to enumerate first-order models. If interpreted theories are added, then even refutationally complete validity checking becomes intractable (linear integer arithmetic with free symbols has a Π^1_1-hard validity problem [Dow72, Hal91]). In practice, this lack of completeness is a major concern in software verification applications, including ranking function and loop invariant synthesis, which require the capability to disprove non-valid proof obligations. In such cases, incomplete theorem provers run out of resources or report 'unknown' instead of detecting non-validity (i. e. , satisfiability of the negated conjecture).

There are various methods to circumvent this problem: SMT-solvers generally use instantiation heuristics to reduce the input problem to a quantifier-free one, while approaches based on first-order theorem proving either are incomplete; do not accept free BG-sorted operators at all, for example [KV07, Rüm08, GK06, BT11]; or, otherwise, are complete only for certain fragments of the input language.

Nieuwenhuis et al. [NOT06] gives an overview of SMT instantiation heuristics, while specific ones are described by Ge et al. [GBT07], and de Moura and Bjorner [dMB07]. These heuristics are complete only in rather restricted cases, as in Ge and de Moura [GdM09]. For theorem proving, approaches described in [BGW94, AKW09, KW12, BW13a, BW13b] all restrict the input language to obtain completeness.

Some complete fragments can be very useful, for example, the data structure theories given previously are known to have finite saturations under the Superposition calculus (when the conjecture is ground and without interpreted theories) [ABRS09]. It seems straightforward to include theory reasoning in these fragments, so long as compactness is not a problem. Since the only new inferences on the BG part of clauses are simplifications or constraint refutations, a finite saturation should be possible. The Define rule is then able to recover sufficient completeness by renaming each of the finitely-many ground free BG-sorted terms in the finite saturation.

More general fragments, such as the array property fragment, allow limited use of quantifiers. These are usually instantiated first, then the proof goal is discharged using a dedicated decision procedure for the ground fragment. Solvers for first-order logic typically degrade in performance as the number of clauses increases, hence it is desirable to minimize the number of instances, if possible. However, their ability to reason natively with quantifiers properly extends the capability of SMT solvers.

As described in Chapter 2, the Hierarchic Superposition calculus requires both compactness of the base specification and sufficient completeness of the input clause set, for refutation completeness. A lack of sufficient completeness either results in non-termination, or, more seriously, termination with a saturated clause set none of whose models properly extend any model of the base specification $\mathcal{B}$. Then, any

clause set that has a finite saturation under the Hierarchic Superposition calculus requires sufficient completeness in order to conclude $\mathcal{B}$-satisfiability.

The GBT-fragment, in which all free BG-sorted terms are ground, is sufficiently complete. This will be the starting point of the method described in this chapter.

The GBT-fragment will be modelled by *finitely quantified clauses*, in which every variable occurring below a free BG-sorted operator is quantified over a finite cardinality subset of its domain. The advantage of this is twofold: instantiation is limited to only those quantifiers which must be instantiated for completeness, and, sets of clause instances (and hence sets of *relevant* terms) can be represented efficiently by $\Sigma_{\mathbb{Z}}$-formulas.

If *all* quantifiers range over finite sets, decidability can be recovered trivially by exhaustive instantiation, followed by calling a suitable SMT-solver. Of course, the instantiation approach scales poorly with increasing domain size, as observed in the context of finite-model finding, for example see [Sla92, ZZ95, McC03, CS03, BFdNT09, RTG+13, RTGK13].

Then the main goal is to design a procedure that recovers sufficient completeness while minimizing instantiation of clauses. To this end, the satisfiability procedure maps multiple free BG-sorted terms to the same constant, and refinements are made by exempting selected terms from that default assignment in a conflict-guided way. After each refinement, the given clause set is rewritten with the new assignment into a clause set with sufficient completeness, so $\mathcal{B}$-satisfiability can be checked with existing reasoners. Suitable reasoners are, e. g. , theorem provers implementing Hierarchic Superposition and, with one more simple transformation step, SMT-solvers for the EA-fragment of the background theory. The procedure stops after finitely many refinement steps; either with a representation of a model (i. e. , a saturated clause set) or a set of clause instances which demonstrates the unsatisfiability of the input clause set.

The satisfiability procedure can be understood as testing a succession of over and under-approximations of the given clause set. Under-approximations are created using a conjectured equality relation on the free BG-sorted terms. Concretely, terms assigned the same default constant are in the same equivalence class. Again, simplifying the clause set using this relation (i. e. , replacing free BG-sorted terms with constants) produces a clause set for which saturation in the Hierarchic Superposition calculus implies $\mathcal{B}$-satisfiability. It is called an under-approximation in keeping with naming conventions, e. g. , in counter-example guided abstraction refinement, where an under-approximation may exclude some Σ-interpretations, but satisfiability of the under-approximation implies satisfaction of the original set.

The over-approximation phase takes a certain subset of clause instances which have been produced by a sound assignment to free BG-sorted terms, and tests this for unsatisfiability. If neither test is conclusive, then the current equality relation is refined by removing some terms from equivalence classes. Effectively, this enlarges the set of Σ-interpretations considered in the under-approximation phase. Doing so naïvely will require more work than simply instantiating outright, and so a critical part of the procedure is the heuristic used to choose the terms to be removed from

the equivalence relation and added as instances after an iteration.

In summary, the satisfiability algorithm aims to fix the immediate problem that follows the restriction to the GBT-fragment: the exponential increase in clause numbers due to instantiation with ground free BG-sorted terms. The fix involves representing clause instances symbolically using LIA formulas, then aggressively replacing relevant terms with constants. This unsound step is rectified by heuristic instantiation of clauses which appear to be causing unsatisfiability; a form of conflict-guided instantiation.

5.2 Example Application

Let $\mathcal{N}$ be the following clause set:

$$\begin{array}{lll} (1) & \mathsf{read}(\mathsf{write}(\mathsf{a},i,x),i) \approx x & (4) \quad 1 \leq \mathsf{m} \wedge \mathsf{m} < 1000 \\ (2) & \mathsf{read}(\mathsf{write}(\mathsf{a},i,x),j) \approx \mathsf{read}(\mathsf{a},j) \vee i \approx j & (5) \quad \mathsf{read}(\mathsf{a},\mathsf{m}) < \mathsf{read}(\mathsf{a},\mathsf{m}+1) \\ (3) & \mathsf{read}(\mathsf{a},i) \leq \mathsf{read}(\mathsf{a},j) \vee \neg(i<j) \vee i \notin [1..1000^i] \vee j \notin [1..1000^j] & \end{array}$$

where $x \in [l..h]$ abbreviates the formula $l \leq x \wedge x \leq h$ for $l, h \in \mathbb{Z}$, and $x \in \chi_{\mathbb{Z}}$.

Notice that (1) and (2) are the axioms for non-extensional, integer-sorted arrays with integer indices, as introduced previously. Axiom (3) states that the array a is sorted within the domain $[1..1000]$ for i and j. Annotating the upper bounds as 1000^i and 1000^j facilitates replacing them with different values for a given variable. The clauses of (4) constrain the integer constant m to the stated range. The goal is to confirm that $\mathcal{N}$ is $\mathcal{T}_{\mathbb{Z}}$-satisfiable.

In the example, sufficient completeness means that in every model of (1)-(5) w. r. t. pure first-order logic, every ground read-term must be equal to some concrete integer. Every write-term inside of a read-term can be eliminated with the axioms (1) and (2). The only problematic terms are applications of read to the array constant a. The clauses (3) and (5) constrain the interpretation of terms of the form $\mathsf{read}(\mathsf{a},t)$ but *do not* enforce sufficient completeness. Achieving sufficient completeness for *ground* clauses like (5) is easy: one just needs to add clauses defining free BG-sorted terms: (5b) $\mathsf{read}(\mathsf{a},\mathsf{m}) \approx \mathsf{n}_0$ and (5c) $\mathsf{read}(\mathsf{a},\mathsf{m}+1) \approx \mathsf{n}_1$ where n_0 and n_1 are fresh integer-sorted parameters, then replace the clause (5) by (5a) $\mathsf{n}_0 < \mathsf{n}_1$. This is akin to the Define rule described in Chapter 2.

The more difficult part concerns the non-ground clause (3). The method of this section generalizes the action of the Define rule by creating definitions for non-ground clauses of the form described in Section 5.3. It begins with a *default assignment* that maps all read-terms of a particular shape to the *same* arbitrary symbolic constant. Applied to clause (3) this produces:

$$\begin{array}{lll} (3a) & \mathsf{n}_3 \leq \mathsf{n}_4 \vee \neg(i<j) \vee i \notin [1..1000^i] \vee j \notin [1..1000^j] & \\ (3b) & \mathsf{read}(\mathsf{a},i) \approx \mathsf{n}_3 \vee i \notin [1..1000^i] & (3c) \quad \mathsf{read}(\mathsf{a},j) \approx \mathsf{n}_4 \vee j \notin [1..1000^j] \end{array}$$

Clauses (3b) and (3c) are the definitions for the default interpretation, one per occur-

rence of a read-term in (3), and clause (3a) is clause (3) after applying these definitions.

The new clause set $\mathcal{N}_1 = \{(1), (2), (3a)-(3c), (4), (5a)-(5c)\}$ needs to be checked for satisfiability. As $\mathcal{N}_1$ has sufficient completeness[1], a Hierarchic Superposition solver can be used to show that it is unsatisfiable. (Alternatively, one can remove all occurrences of the read-operator in the clauses (3a)-(5c) by exhaustive Superposition-like inferences, and then submit the resulting clause set to a suitable SMT-solver).

The unsatisfiability of $\mathcal{N}_1$ implies that $\mathcal{N}$ is not satisfied using the current constraints on the interpretation of read (i. e., definitions), however, it may be satisfied by less strict constraints. The next step is to refine the default interpretation specified by clauses (3a), (3b), (3c), at a critical point that is responsible for unsatisfiability. The heuristic, described in Section 4.2, determines that point by first finding a maximal sub-domain for which the clause set is satisfiable. In the example, this is the sub-domain $[1..999^i]$ for the variable i and the point is 1000. Specifically, the set $\mathcal{N}_2$ obtained from $\mathcal{N}_1$ by replacing 999^i by 1000^i everywhere is satisfiable. The refinement is made by excluding the point 1000 from the default interpretation and providing a separate definition for it:

$$
\begin{array}{ll}
(3a1) & \mathsf{n}_{31} \leq \mathsf{n}_4 \vee \neg(i < j) \vee i \notin [1..1000^i] \setminus \{1000\} \vee j \notin [1..1000^j] \\
(3a2) & \mathsf{n}_{32} \leq \mathsf{n}_4 \vee \neg(1000 < j) \vee j \notin [1..1000^j] \\
(3b1) & \mathsf{read}(\mathsf{a}, i) \approx \mathsf{n}_{31} \vee i \notin [1..1000^i] \setminus \{1000\} \\
(3b2) & \mathsf{read}(\mathsf{a}, 1000) \approx \mathsf{n}_{32} \\
(3c) & \mathsf{read}(\mathsf{a}, j) \approx \mathsf{n}_4 \vee j \notin [1..1000^j]
\end{array}
$$

Clauses (3b1) and (3b2) provide the modified definitions, and clauses (3a1) and (3a2) are the rewritten versions of (3). Let $\mathcal{N}_3 = \{(1), (2), (3a1) - (3c), (4), (5a) - (5c)\}$ be the result of the current transformation step; it remains unsatisfiable. In the next round, the new upper bounds defining the satisfiable subset of $\mathcal{N}_3$ are 999^j and 1000^i. Transforming clause (3) w. r. t. the points 1000 for j and 1000 for i from the previous step gives:

$$
\begin{array}{ll}
(3a1) & \mathsf{n}_{31} \leq \mathsf{n}_{41} \vee \neg(i < j) \vee i \notin [1..1000^i] \setminus \{1000\} \vee j \notin [1..1000^j] \setminus \{1000\} \\
(3a2) & \mathsf{n}_{32} \leq \mathsf{n}_{41} \vee \neg(1000 < j) \vee j \notin [1..1000^j] \setminus \{1000\} \\
(3a3) & \mathsf{n}_{31} \leq \mathsf{n}_{42} \vee \neg(i < 1000) \vee i \notin [1..1000^j] \setminus \{1000\} \\
(3a4) & \mathsf{n}_{32} \leq \mathsf{n}_{42} \vee \neg(1000 < 1000)
\end{array}
$$

$$
\begin{array}{llll}
(3b1) & \mathsf{read}(\mathsf{a}, i) \approx \mathsf{n}_{31} \vee i \notin [1..1000^i] \setminus \{1000\} & (3b2) & \mathsf{read}(\mathsf{a}, 1000) \approx \mathsf{n}_{32} \\
(3c1) & \mathsf{read}(\mathsf{a}, j) \approx \mathsf{n}_{41} \vee j \notin [1..1000^j] \setminus \{1000\} & (3c2) & \mathsf{read}(\mathsf{a}, 1000) \approx \mathsf{n}_{42}
\end{array}
$$

Let $\mathcal{N}_4 = \{(1), (2), (3a1) - (3c2), (4), (5a) - (5c)\}$ be the result of the current transformation step. This time, $\mathcal{N}_4$ is satisfiable, and so is $\mathcal{N}$, with the same models. If I is any such model, we have $I(\mathsf{m}) = 999$, $I(\mathsf{read}(\mathsf{a}, i)) = k$, for some integer k and all $i = 1..999$, and $I(\mathsf{read}(\mathsf{a}, 1000)) = l$ for some integer $l > k$. The reasoning behind this procedure is formalized in Section 5.4.

[1] or an approximation thereof– see later

The example is solved after two iterations of transformation steps. In general, each transformation step needs $O(m \cdot \log(n))$ prover calls to determine the next point as explained above, where m is the number of FQ variables in the given clause set and n is the size of the largest domain. With $m = 2$ and $n = 1000$, this accounts for $2 \cdot (m \cdot \log(n)) \leq 40$ theorem prover calls, however, each one is rather simple. In contrast, the full ground instantiation of the clauses (3)-(5) has a size of $n^m = 10^6$, which is far too large for current theorem provers or SMT-solvers.

When every default assignment is unsuitable, the given method also requires a full ground instantiation, as separate definitions are needed for each term instance in order to establish overall (un)satisfiability. Unfortunately, the naïve heuristic presented in the example only permits *single* exception points to be added at each step. So not only is the fully instantiated clause set checked, but also every step-wise refinement on the way to reaching it. That is, one transformation step for each individual domain element followed by a prover run on the clause set instantiated over all finite quantifier domains.

A section in the next chapter will show how to identify clause sets which necessarily have this behaviour, and how to avoid them with a syntactic check. A specialized representation of introduced definitions for free BG-sorted terms is also given, which allows finding *ranges* for exceptions rather than just single points.

5.3 Finite Cardinality Theories

This section describes a general theory of finite structures and some reasoning methods over them. A definition for *Finitely Quantified (FQ)-clauses* over integers gives a specific fragment for modelling the GBT-fragment. Then, a transformation from general theories which define finite sets (or possibly finite sorts) into sets of FQ-clauses allows reasoning over larger fragments. These will form the basis for both of the refinement algorithms presented in the current and the following chapter.

Definition 5.3.1 (Cardinality Constraint Clause)**.** The cardinality of the domain of interpretations can be bounded using *cardinality constraint clauses*:

$$x \approx c_1 \vee \ldots \vee x \approx c_n$$

where for each $1 \leq i \leq n$, c_i is a distinct constant.

Any model of a cardinality constraint clause with n constants can have at most n distinct elements in its domain. In a many-sorted language such a clause would bound the cardinality of the carrier set for the sort of x in any model.

Example 5.3.1. Enumerated data-types (e. g. , Booleans) and data-types implemented with fixed-width bit-vectors like char can be modelled with cardinality constraint clauses.

Unfortunately, the rules of the Superposition calculus allow self-inferences on

constraint clauses:

$$\frac{x \approx c_1 \vee \ldots \vee x \approx c_n \qquad y \approx c_1 \vee \ldots \vee y \approx c_n}{x \approx y \vee x \approx c_2 \vee \ldots \vee y \approx c_n}$$

These produce many, mostly unhelpful, clauses. Such behaviour is noted by Hillenbrand and Weidenbach [HW07] as the motivation for their calculus: an adaptation of the Superposition calculus to the case where *all* sorts are bounded by cardinality clause constraints. Although it seems evident that the Superposition calculus should decide this theory (by grounding all clauses first, then by decidability of completion for ground finite rewrite systems), their goal is to describe an efficient calculus for this theory. Although decidability was shown in principle, the calculus enumerates exponentially many interpretations for uninterpreted functions and also contains a computationally difficult check for redundancy of inferences.

An alternative could be to use the Hierarchic Superposition calculus with a dedicated solver for finite cardinality theories, such as the solver described by Reynolds et al. [RTGK13]. The solver could then delete tautologies like those (eventually) produced by self-inferences with cardinality constraint clauses. However, similar clauses result from inferences like the above where x is replaced with a foreground term. As these are impure (containing a mix of FG and BG terms), they are not removed by simplification and remain eligible for other inferences. Then the finite domain solver will need to test impure clauses as well. Such checks are more expensive, since each mixed clause must be tested independently, i. e. , incremental model finding cannot be used. If this is done frequently– as it must be for simplification to be effective– performance will be severely impeded, as the satisfiability check for finite cardinality theories is NP-complete.

If roles are reversed and a finite model is chosen *before* the proof, then simplification checks amount to testing whether a given BG clause is true in the selected model. The model can also be used to simplify BG literals in impure clauses, but if any pure BG clauses are not satisfied in that particular model, a new model must be chosen. Effectively, this pushes the model search out from between inferences to between derivations. This is the general approach of the refinement procedure, except that the default interpretation constrains the possible models rather than explicitly giving a model.

One could find a similar philosophy in the AVATAR system for first-order reasoners described by Voronkov [VB14]. It uses a SAT solver to choose a maximal splitting of clause components before a proof, effectively fixing some part of the interpretation then using a Superposition-based solver to work out the details. This split is refined when evidence of its unfeasibility is found, similar to the above. This comparison holds only at a high level: unlike AVATAR this method is focused just on the problem of reasoning in combinations of theories.

5.3.1 Finitely Quantified Clauses

The following assumptions on clauses guarantee that there are finitely many free BG-sorted term instances among the ground instances of a clause set:

- All subterms headed by a BG-sorted foreground (BSFG) operator have only $\mathbb{Z}$-sorted free variables, and
- These variables are quantified over finite (integer) sets.

The second assumption could be weakened to allow variables of any sort whose cardinality is restricted by a cardinality constraint by giving a map from that sort to a suitable subset of integers, see Section 5.3.2.

Let $\xi \in \Xi_B$ be a BG sort. A *finite ξ-domain* Δ is any, possibly empty, finite set $\{d_1, \ldots, d_n\} \subseteq Dom(\Sigma_B)$ of ξ-sorted domain elements d_i. Membership in Δ can be expressed by a Σ_B-formula $\mathcal{F}_\Delta[x]$ in one free ξ-sorted variable x whose extension is exactly the set Δ, in every $\mathcal{B}$-interpretation[2]. A finite set Δ can always be represented as a disjunction: $\mathcal{F}_\Delta[x] = x \approx d_1 \vee \cdots \vee x \approx d_n$. However, the formula $\mathcal{F}_\Delta[x]$ is intended to be used as a guard for a regular clause, i. e., $\mathcal{F}_\Delta[x] \Rightarrow C[x]$, this will be translated by a solver to the CNF form $C[d_1] \wedge \ldots \wedge C[d_n]$. As mentioned above, a critical factor in the choice to model finite domains with sets of integers is the fact that finite sets can be compactly described by $\Sigma_\mathbb{Z}$-formulas:

Definition 5.3.2 (Domain Formula)**.** A finite $\mathbb{Z}$-domain $\Delta = \{d_1, \ldots, d_n\}$ with minimal and maximal elements d_{min}, d_{max} respectively, can be represented by either of the formulas

$$d_{min} \leq x \wedge x \leq d_{max} \wedge \bigwedge_{c \in S} x \not\approx c \tag{5.1}$$

$$x \approx d_1 \vee \cdots \vee x \approx d_n \tag{5.2}$$

where $S = [d_{min}, d_{max}] \setminus \Delta$. These are called *domain formulas* for Δ.

Using (5.1) requires the background domain to have a non-dense partial order, which essentially restricts usage to the integers. Note that the latter part of the formula ($\bigwedge_{c \in S} x \not\approx c$) could also be considered part of the clause, but generally it pays for the instantiation procedure to have the most specific representation possible of the finite domain.

In the following, domain formulas will be abbreviated with set-like notation: $x \in \Delta$ or $x \in [0, 100]$. In particular, the form $x \in \Delta \setminus \Pi$ is used to distinguish certain domain elements Π that are excluded from Δ, although $x \in \Delta \wedge \neg(x \in \Pi)$ is itself a domain formula. Where a domain formula includes only a single free variable, that is indicated with a subscript, e. g., Δ_x could refer to the previous formula. Domain formulas can also appear as literals in clauses, usually negated, e. g., $x \notin \Delta$. This is only at the outer-loop level; all domain formulas inside clauses are expanded to their full CNF equivalents before being passed to a solver.

[2] specifically $\mathcal{F}_\Delta[x]$ must not contain parameters

By guarding all variables in a clause that occur below a free BG-sorted operator with domain formulas, the size of the set of relevant terms can be restricted to be finite.

Definition 5.3.3 (Finitely Quantified Clause)**.** A *finitely quantified clause* is a Σ-clause of the form $D \vee x_1 \notin \Delta_{x_1} \vee \cdots \vee x_n \notin \Delta_{x_n}$, where $n \geq 0$, such that

1. $x_i \neq x_j$ for $1 \leq i < j \leq n$, and
2. every variable occurring below a free BG-sorted operator in D is in $x_1, \ldots, x_n$.

Let $\mathsf{FQvars}(C \vee \neg\Delta)$[3] be the set of variables of C which appear in Δ.

Example 5.3.2. The following are finitely quantified clauses:

$$(C_1) \quad f(x_1) > x_1 + y \vee \neg(y > 0) \vee x_1 \notin [1..1000]$$
$$(C_2) \quad f(x_2 + g(x_3)) < 10 \vee \neg(x_2 > 2) \vee x_2 \notin [1..1000] \vee x_3 \notin [1..100]$$

In C_1 the variable y does not need to be guarded by a domain formula, as it does not occur below a free BG-sorted operator. The literal $x_1 \notin [1..1000]$ abbreviates the negated domain formula $\neg(1 \leq x_1 \wedge x_1 \leq 1000)$, similarly for $x_2 \notin [1..1000]$. Extra Σ_B-literals such as $\neg(x_2 > 2)$ are typically not in the domain formula; regardless, the existence of a domain formula guarantees that x_2 can take only finitely many values.

5.3.2 Indexing Finite Sorts

FQ-clauses require all variables below free BG-sorted terms to be integer sorted and restricted to finite domains. This section gives a possible way of lifting the restriction to integer sorted variables, by means of a transformation from clause sets with free BG-sorted terms that include variables ranging over arbitrary finite domains to FQ-clause sets. It describes sort encoding similar to those in [HW07, HW13]. There are two ways to restrict quantifiers to a finite domain: by *cardinality constraint clauses*, or by restriction using the base specification. Both situations can be modelled using FQ-clauses, by introducing a map from the finite set to integers.

5.3.2.1 Finite Predicates

Consider the problem of searching for a counter-example element among a finite subset of a sort, for example, a list of length three containing characters (i. e. , an integer in the range $[0, 255]$). The *LIST* sort cannot be restricted to only contain lists of length three, since the list constructor axiom allows constructing lists of any length. Also, the many-sorted signature does not permit distinct sorts to have a non-empty intersection. Instead, the domain of interest is defined using a new predicate, e. g. ,

$$dom(x) \Leftrightarrow x \approx \mathsf{nil} \vee x \approx \mathsf{cons}(a_{11}, \mathsf{nil}) \vee \ldots \vee x \approx \mathsf{cons}(a_{31}, \mathsf{cons}(a_{32}, \mathsf{cons}(a_{33}, \mathsf{nil})))$$

[3] $\Delta = x \in \Delta_x \wedge y \in \Delta_y \wedge \ldots$

This can be used to guard *LIST*-sorted quantifiers in the problem specification.

As for FQ-clauses, all variables below BSFG operators must range over finite sets, only now domain formulas are replaced with cardinality constraint clauses over arbitrary sorts. It is assumed that cardinality constraint clauses have predicates of the form Card_k as aliases:

$$\mathsf{Card}_k(x) \Leftrightarrow x \approx d_1 \vee \ldots \vee x \approx d_k$$

where the d_i terms are ground and pairwise distinct. The predicate Card_k can only be interpreted as a set with *at most k* different values. Then, a *finitely bounded* (FB)-clause is such that any occurrence of a variable in a free BG-sorted term is guarded by a predicate Card_k of appropriate sort. As a consequence, only variables appear as arguments to cardinality predicates.

The idx transformation from FB-clause sets to FQ-clause sets is:

Definition 5.3.4. For each predicate $\mathsf{Card}_k : S \to \mathsf{Bool}$:

1. Add a new operator $i_S : \mathbb{Z} \to S$ to Σ
2. Replace each FB-clause $C[t[x]] \vee \neg\mathsf{Card}_k(x)$, where t is a free BG-sorted term, with the FQ-clause $C[x/i_S(y)] \vee y \notin [1,k]$
3. Replace $\mathsf{Card}_k(x) \Leftrightarrow x \approx d_1 \vee \ldots \vee x \approx d_k$ with $i_S(1) \approx d_1 \wedge \ldots \wedge i_S(k) \approx d_k$.

For a clause set $\mathcal{N}$ containing Card_k predicates and definitions, let $\mathsf{idx}(\mathcal{N})$ be the result of applying the above for each predicate Card_k.

Lemma 5.3.1. *Let $\mathcal{N}$ be a set of FB-clauses over signature Σ that includes definitions for cardinality predicates over sorts $\{S_1, \ldots, S_n\}$. Then $\mathcal{N}$ and $\mathsf{idx}(\mathcal{N})$ are equisatisfiable over the extended signature $\Sigma \cup \{i_S : S \in \{S_1, \ldots, S_n\}\}$*

Proof. $\Leftarrow$: Assume $I \models \mathcal{N}$. For each cardinality constraint Card_k there is a finite set $C^I = \{d \in \mathcal{D}_I : I \models \mathsf{Card}_k(d)\}$. Define an arbitrary enumeration of the elements of C^I, i. e., let $C^I = \{d_1, \ldots, d_k\}$. For each cardinality predicate $\mathsf{Card}_k : S \to \mathsf{Bool}$ define $i_S^I : \mathbb{Z} \to S$ as

$$i_S^I(x) \begin{cases} d_x & \text{if } 1 \leq x \leq k \\ d_1 & \text{otherwise} \end{cases}$$

This satisfies each of the clauses $C[t[i_S(x)]] \vee x \notin [1,k]$, since the argument to i_S is guarded by a domain formula and the clause is satisfiable for each of the instances that satisfy the guard.

$\Rightarrow$: Assume $\mathsf{idx}(\mathcal{N})$ has a model I. Define a new Σ-interpretation I' such that $\mathsf{Card}_k^{I'}$ is the set $\{I(i_S(1)), \ldots, I(i_S(k))\} = \{d_1, \ldots, d_k\}$. The only clauses of $\mathcal{N}$ not already satisfied by I are those with a Card_k predicate. Clearly $I' \models \mathsf{Card}_k(x) \Leftrightarrow x \approx d_1 \vee \ldots \vee x \approx d_k$. The remaining clauses to satisfy have the form $C[t[y]] \vee \neg\mathsf{Card}_k(y)$. By assumption, $I' \models C[t[d_j]]$ for $j \in [1,k]$, and so $I' \models \forall y.\, C[t[y]] \vee \neg\mathsf{Card}_k(y)$. □

5.3.2.2 Finite Sorts

The restriction to a finite cardinality can also be modelled via specifications in which certain base sorts have a restricted cardinality. Given sorts Ξ in Σ, a *cardinality map* over Σ is $\mathsf{card} : \Xi \to \mathbb{N} \cup \{\infty\}$, where for all S $\mathsf{card}(S) \geq 1$. For a cardinality map card over Σ, a cardinality bounded specification is a specification in which all interpretations in the model class of the specification have *at most* $\mathsf{card}(S)$ elements in the carrier set for S, where $\mathsf{card}(S) \neq \infty$.

As for FQ-clauses, all variables below BSFG operators must range over finite sets, only now that restriction is expressed by the cardinality map and enforced by the bounded specification. In this context an FB-clause is such that for any BG-sorted non-base subterm $t[x]$, if $\mathrm{sort}(x) = S$, then $\mathsf{card}(S) \neq \infty$.
The transformation idx from FB-clause sets over bounded specifications to FQ-clause sets is:

Definition 5.3.5. For each sort S where $\mathsf{card}(S) \neq \infty$

1. Add a new operator $i_S : \mathbb{Z} \to S$ to Σ
2. Replace each clause $C[t[x]]$, where t is a free BG-sorted term and $\mathrm{sort}(x) = S$, with the FQ-clause $C[x/i_S(y)] \vee y \notin [1, \mathsf{card}(S)]$
3. For each constant c of sort S add the clauses[4]
   - $i_S(\beta_c) \approx c$
   - $1 \leq \beta_c$
   - $\beta_c \leq \mathsf{card}(S)$

   for fresh parameter β_c
4. For every function symbol $f : S_1 \times \ldots \times S_k \to S$ where S is finitely bounded, add the clause $f(x_1, \ldots, x_k) \approx i_S(1) \vee \ldots \vee f(x_1, \ldots, x_k) \approx i_S(\mathsf{card}(S))$

Steps 1 and 2 are the same as for cardinality bounding predicates, while steps 3 and 4 extend the cardinality bound to operator symbols of the appropriate sort.

Lemma 5.3.2. *For an FB-clause set $\mathcal{N}$, $\mathsf{idx}(\mathcal{N})$ is equisatisfiable with $\mathcal{N}$, and $\mathsf{idx}(\mathcal{N})$ is an FQ-clause set.*

Proof. $\Leftarrow$: Assume $\mathcal{N}$ has a model M respecting the cardinality restriction card. For a finitely bounded sort S define an arbitrary enumeration of the elements of S^M (i. e., the carrier set for S), so $S^M = \{s_1, \ldots, s_k\}$ such that $k \leq \mathsf{card}(S)$. For each finitely bounded S define $i_S^M : \mathbb{Z} \to S$ as

$$i_S^M(x) \begin{cases} s_x & \text{if } 1 \leq x \leq \mathsf{card}(S) \\ s_1 & \text{otherwise} \end{cases}$$

[4] If there are less than $\mathsf{card}(S)$ constants, then assign to each existing constant c in S an arbitrary unique integer k where $1 \leq k \leq \mathsf{card}(S)$, and add the clause $i_S(k) \approx c$.

This satisfies each of the existential clauses in $\mathsf{idx}(\mathcal{N})$ introduced for constants, and each of the clauses $C[t[i_S(x)]] \vee \neg x \in \Delta$, since the argument to i_S is guarded and the clause is satisfiable for each of the instances that satisfy the guard.

$\Rightarrow$: Assume $\mathsf{idx}(\mathcal{N})$ has a model M. The model M respects the given cardinality bounds if it satisfies, for each S,

$$\forall x : S.\ \exists y : \mathbb{Z}.\ 1 \leq y \leq \mathsf{card}(S) \wedge x \approx i_S(y)$$

This is ensured by the constraints added for function and constant symbols in steps 3, 4 of the idx transform. So $M \models \mathcal{N}$. □

Lemma 5.3.3. *For constants c_i and variable x of sort S, where $\mathsf{card}(S) = n$; the cardinality constraint clause $x \approx c_1 \vee \ldots \vee x \approx c_n \in \mathcal{N}$ is redundant w.r.t. $\mathsf{idx}(\mathcal{N})$.*

Although the above seems almost tautological, it allows eliminating cardinality constraint clauses using the transform idx. By taking the presence of the cardinality constraint to mean $\mathsf{card}(S) = n$ in the base specification, the over productive cardinality clause can be dropped.

Example 5.3.3. This can be used to do a form of finite model finding on arbitrary formulas with BSFG operators by guarding all variables below BSFG operators with arbitrarily chosen cardinality constraint predicates. A saturation will imply a model, but a contradiction will require the constraints to be enlarged.

This is similar, in spirit, to the algorithm applied to FQ-clauses, although more inefficient as the runs are independent. Hence, the next chapter introduces specialized algorithms for T_{LIST} and other recursive data structure theories, that exploit the structure of data structures.

5.4 Domain-First Search

This section describes the algorithm checkSAT defined in Figure 5.4. It is based on the algorithm in [BBW14], and formalizes the example in Section 5.2. The aim of checkSAT is to show $\mathcal{B}$-satisfiability of a set of FQ-clauses, while producing a minimal number of clause instances. As it uses the finite domains of the given FQ-clause set to organize the clause instances, it is described as *domain-first search*.

The main advantage of this version of checkSAT is the fact that the entire algorithm (both checkSAT and find) can be implemented using off-the-shelf solvers, unlike the version in the next chapter.

Internally checkSAT allows finite domains to be shared between clauses: a set of FQ-clauses $\{C_1 \vee \neg\Delta_1, \ldots, C_n \vee \neg\Delta_n\}$ is represented by the formula $(\Delta_1 \wedge \ldots \wedge \Delta_n) \Rightarrow (C_1 \wedge \ldots \wedge C_n)$ as in line 2. The domain formulas in the antecedent are called *global* domain formulas. The sets of excluded points Π_x at line 10 are emphasized, as they track progress in the algorithm.

The algorithm does not specifically require the input clause set to be variable disjoint. In fact the algorithm performs differently depending on which finite domains

```
1  algorithm checkSAT((Δ_{x_1} ∧ ... ∧ Δ_{x_k}) ⇒ (C_1 ∧ ... ∧ C_n))
2  // returns 'B-satisfiable' or 'B-unsatisfiable'
3  let M = (Δ_{x_1} \ ∅ ∧ ... ∧ Δ_{x_k} \ ∅) ⇒ (C_1 ∧ ... ∧ C_n)
4  while true
5   let M⁻ = definitional(M) //see Section 5.4.1
6   let M⁺ = persistent(M⁻) //see Section 5.4.1
7   if M⁻ is satisfiable return B-satisfiable // justified by Lemma 5.4.2
8   if M⁺ is B-unsatisfiable return B-unsatisfiable
9   let (x, d) = find(M)
10  M := (Δ_{x_1} \ Π_{x_1} ∧ ... ∧ Δ_x \ (Π_x ∪ {d}) ∧ ...) ⇒
11          (C_1 ∧ ... ∧ C_n) ∧ (C_1 ∧ ... ∧ C_n)[x/d]
```

Figure 5.1: The algorithm for hierarchic satisfiability

are shared between clauses. There are two extremes of domain sharing that can occur: at one end, every clause is variable disjoint from all others. This degrades performance both in `find` which can require as many prover calls as there are (global) finite domains, and also where variants of relevant terms under different domains cancel the effect of exceptions (see Example 5.7.1). The opposite extreme is to identify finitely quantified variables in different clauses according in some fixed order so that there are only as many global finite domains as the maximal number of finitely quantified variables in any individual clause. The disadvantage is that any change in a finite domain affects every clause, adding a new instance of each finitely quantified clause at each iteration. Actual performance changes realized, depend heavily on the particular clause set, hence the final choice is left with the user.

checkSAT repeatedly applies a transformation of the formula $(\Delta_{x_1} \setminus \Pi_{x_1} \wedge \ldots \wedge \Delta_{x_k} \setminus \Pi_{x_k}) \Rightarrow (C_1 \wedge \ldots \wedge C_n)$ to an equisatisfiable set of FQ-clauses w. r. t. growing sets of exception points Π_x. It is assumed that each $\Pi_x \subseteq \Delta_x$. If $\Pi_x = \Delta_x$, then Δ_x is tacitly removed from the set of global domain formulas to avoid a tautology. Note that $\Delta_x \setminus \Pi_x$ is also a domain formula, specifically $(d_1 \leq x \wedge x \leq d_2 \wedge \bigwedge_{e \in S} x \not\approx e) \wedge \bigwedge_{e \in \Pi_x} x \not\approx e$ has form (1) in Definition 5.3.2. So the exception points are usually implicit, except in the context of the `checkSAT` algorithm.

The procedure stops if any transformed clause set is either $\mathcal{B}$-satisfiable or serves to demonstrate $\mathcal{B}$-unsatisfiability. It is assumed that $\mathcal{B}$-satisfiability tests, i. e., lines 7 and 8, carried out by checkSAT are effective. This is always the case when there are no FG operators other than free BG-sorted operators and the EA-fragment of the background theory is decidable, for example.

If the clause set is unsatisfiable w. r. t. the current set of exception points, then a new exception point is found using the heuristic find. To prove termination of checkSAT, it is enough to choose any $d \notin \Pi_x$. However, choosing arbitrarily can lead to worse overall performance than simply instantiating the FQ-clauses immediately.

The limit of this process of adding exception points is the clause set with all finite quantifiers instantiated over their domains. As shown by Lemma 5.4.4, this is a sound transformation of the clause set. Hence the essential property of checkSAT is

Theorem 5.4.1 (Correctness of checkSAT)**.** *For any set $\mathcal{N}$ of FQ-clauses,* checkSAT$(\mathcal{N})$ *terminates with the correct result: '$\mathcal{B}$-satisfiable' or '$\mathcal{B}$-unsatisfiable'. Moreover, if the result is '$\mathcal{B}$-unsatisfiable' then* checkSAT(N) *with all domain formulas removed is $\mathcal{B}$-unsatisfiable.*

Proof. Termination follows from the fact that find always returns some pair (x, d) such that $x \in \mathbf{x}$ and $d \in \Delta_x \setminus \Pi_x$, as shown in Lemma 5.4.6. Hence, the set Π_x grows monotonically in line 10 in checkSAT, and there are only finitely many elements in Δ_x available for that. Correctness follows from the lemmas in the following section: If checkSAT reports '$\mathcal{B}$-unsatisfiable', then M^+ is unsatisfiable at line 8. By Lemma 5.4.5, this is because M itself is $\mathcal{B}$-unsatisfiable. If checkSAT reports '$\mathcal{B}$-satisfiable', then M^+ is satisfiable at line 7. By Theorem 5.4.2, M^+ is $\mathcal{B}$-satisfiable, and by Lemma 5.4.3 it follows that $\mathcal{N}$ is $\mathcal{B}$-satisfiable and with the same model. The end result of the instantiation at line 10 is a sound instantiation of $\mathcal{N}$, by Lemma 5.4.4. In that case, M^+ and M^- are identical and the result is true of $\mathcal{N}$ in any case. □

In summary, checkSAT tests the $\mathcal{B}$-satisfiability of an input set of FQ-clauses relative to a *subset* of possible models that is specified by the exception points. The constraints are progressively weakened in an attempt to limit instantiation. This weakening is informed by a heuristic computed from the current formula set. Although testing subsets of models is unsound, all models are eventually tested and so the method is sound overall.

5.4.1 Clause Set Approximations

The suggestively named clause sets M^+ and M^- in checkSAT in Figure 5.4 are over and under-approximations of M respectively, as evidenced by the satisfiability status they confer on lines 7 and 8. If the over-approximation M^+ is unsatisfiable, then M is also, and satisfiability of the under-approximation M^- implies satisfiability of M. The converse does not hold in either case.

The clause set M^- is produced by replacing all free BG subterms t of M in innermost-first order with fresh parameters α, and then adding the *definition* unit clause $t \approx \alpha$. Specifically, a definition in this context is an equation of the form $f(t_1, \ldots, t_k) \approx \alpha$, where f is a BSFG operator and α is a parameter which does not appear in the original clause set. FQ-Clauses in which a definition is guarded by a domain formula, e. g., $f(t_1, \ldots, t_k) \approx \alpha \vee \neg\Delta$ are also referred to as definitions, although they really represent sets of definitions.

Since the under-approximation asserts that instances of non-ground free BG subterms under the same definition must be equal, the set of possible models of M is restricted to a subset of models that respect this constraint. Critically, the set M^- has (a version of) sufficient completeness, and so satisfiability of M^- implies $\mathcal{B}$-satisfiability.

The algorithm definitional ensures domain formulas are added to clauses and definitions, and rewrites clauses appropriately. By a slight abuse of notation, Δ and N stand for sets of domain formulas and clauses respectively, so written to make the connection to the formula M clear. A *minimal* free BG term is any free BG term that does not contain a free BG term as a proper subterm. The result of definitional

```
1  algorithm definitional(Δ ⇒ N)
2  let Cls = {C ∨ ¬(⋀_{x ∈ vars(C) ∩ vars(Δ)} Δ_x) : C ∈ N}
3  let Def = ∅
4  while C_i ∈ Cls has minimal free BG subterm t:
5   //α is a fresh parameter
6   Def = Def ∪ {t ≈ α ∨ ¬(⋀_{x ∈ vars(t)} Δ_x)}
7   replace C[t] with C[α] in Cls
8  return Cls ∪ Def
```

Figure 5.2: definitional creates an under-approximation of N using global domain Δ

consists of the sets Cls and Def, such that Cls does not contain any free BG-sorted subterms and all clauses in Def are definition clauses in which all free BG-sorted terms have no proper free BG-sorted subterms.

The following lemma shows that the action of replacing FQ-clauses by their instances does not affect satisfiability.

Lemma 5.4.1. *For any x and $d \in \Delta_x$,*

$$((\Delta_{x_1} \wedge \ldots \wedge \Delta_x \setminus \{d\} \wedge \ldots) \Rightarrow (C_1 \wedge \ldots \wedge C_n)) \wedge (C_1 \wedge \ldots \wedge C_n)[x/d]$$

is equivalent modulo $T_{\mathbb{Z}}$ to $(\Delta_{x_1} \wedge \ldots) \Rightarrow (C_1 \wedge \ldots \wedge C_n)$

Proof. By rewriting with logical equivalences and using $d \in \Delta_x$ to simplify. □

This corresponds to the step on line 10 of the checkSAT algorithm.

Lemma 5.4.2. *Let $\mathcal{N}$ be a set of FQ-clauses with global domains $\Delta \Rightarrow (C_1 \wedge \ldots \wedge C_k)$. If* definitional$(\Delta \Rightarrow (C_1 \wedge \ldots \wedge C_k))$ *is $\mathcal{B}$-satisfiable, then $\mathcal{N}$ is $\mathcal{B}$-satisfiable.*

Proof. Let the result of definitional$(\Delta \Rightarrow (C_1 \wedge \ldots \wedge C_k))$ be Cls ∪ Def and assume that Cls ∪ Def is $\mathcal{B}$-satisfiable. The definitions in Def are exhaustive in the sense that any instance of an FQ-clause C_i obtained by ground instantiation with domain elements of Δ is congruent with some clause in Cls obtained by paramodulation with clauses in Def. This entails that $\mathcal{N}$ ∪ Def is $\mathcal{B}$-satisfiable, and hence, so is $\mathcal{N}$. □

Lemma 5.4.3. *If $M = (\Delta_{x_1} \setminus \Pi_{x_1} \wedge \ldots \wedge \Delta_{x_k} \setminus \Pi_{x_k}) \Rightarrow (C_1 \wedge \ldots \wedge C_n)$ and M^- is $\mathcal{B}$-satisfiable, then M is $\mathcal{B}$-satisfiable for any combination of exception points Π_x.*

Proof. By the previous two lemmas. □

The limit of the process of instantiation in Lemma 5.4.1 is free from domain formulas and includes all instances of $(C_1 \wedge \ldots C_n)$ over the finite domains. Applying definitional to these clauses produces a set Cls ∪ Def where all definitions in Def are ground. As Cls is the result of rewriting ground free BG-sorted terms in each clause with parameters unique to each term instance, it follows that Cls ∪ Def is equivalent to the instances.

Lemma 5.4.4. *If Δ is empty, then* $\mathrm{definitional}(\Delta \Rightarrow N)$ *is equivalent to N.*

So if checkSAT reports unsatisfiable on the final clause set, then the conclusion is sound.

The set of clauses $M^- = \mathrm{Cls} \cup \mathrm{Def}$ does not have sufficient completeness. For example, $\mathrm{Def} = \{f(0) \approx \alpha_0, f(x) \approx \alpha_1 \vee x \notin [0,100] \setminus \{0\}\}$; the instance $f(-1)$ is not assigned a parameter, and so can be interpreted freely by an interpretation.

This problem could be fixed by including, e.g., $f(x) \approx \alpha_2 \vee x \in [0,100]$. In general, for every definition term t include a new definition whose domain formula is built from the negation of all domain formulas found in other definitions for t. Then every simple instance of a free BG-sorted term (and so every relevant term) is defined as equal to some parameter, not just those inside the finite domain. Including such definitions will have no effect on the satisfiability of M^-, as free BG-sorted terms only appear in definitions. In a derivation, the new definitions only ever superpose on existing definitions, which have disjoint domains by construction. Thus, all inferences between an introduced definition and an existing one produce only tautologies.

This appears to be an unnecessary waste of prover effort for the sake of theoretical completeness, since the same interpretation satisfies the clause set without including those extra definitions.

Theorem 5.4.2. *Let $M = \Delta \Rightarrow (C_1 \wedge \ldots \wedge C_n)$ be a set of FQ-clauses as above. If M^- is satisfiable, then it is $\mathcal{B}$-satisfiable.*

Proof. Let $M_2 = \mathrm{vsgi}(M^-) \cup \{t \approx \alpha : \neg\Delta \vee t \approx \alpha \in \mathrm{vsgi}(M^-) \text{ and } T_{\mathbb{Z}} \models \neg\Delta\}$. Every $t \in \mathrm{rel}(M^-)$ is defined in M_2, so it has local sufficient completeness. Moreover, any model of M_2 satisfies M^-, as every new unit clause in M_2 subsumes only trivial clauses in $\mathrm{vsgi}(M^-)$. Assume for contradiction that M_2 is not $\mathcal{B}$-satisfiable. Then there is a derivation of the empty clause from M_2 using the Hierarchic Superposition calculus. This derivation cannot include any new definitions $t \approx \alpha$, because t only occurs in either the definition or a trivial ground clause of $\mathrm{vsgi}(M^-)$, which is immediately redundant. Then the derivation of the empty clause uses premises from $\mathrm{vsgi}(M^-)$ only, contradicting that M^- is satisfiable. Therefore, M_2 is $\mathcal{B}$-satisfiable, and that model satisfies M^-. □

This is enough for completeness of the Hierarchic Superposition calculus on the clause set M^-, as shown in Chapter 2.

The over-approximation M^+ selects a subset of clauses and definitions in M^- which do not contain any variables in the domain formulas Δ. Concretely, given that $M = \Delta \Rightarrow (C_1 \wedge \ldots \wedge C_k)$, define $M^+ = \{C \in M^- : \mathrm{vars}(C) \cap \mathrm{vars}(\Delta) = \emptyset\}$. This set of clauses is equivalent to a subset of the set produced by fully instantiating all finite quantifiers of M. Since this full instantiation is equisatisfiable with M, the unsatisfiability of M^+ implies the unsatisfiability of M.

Lemma 5.4.5. *If M^+ is unsatisfiable, then M is unsatisfiable.*

Proof. Every clause in M^+ is either unchanged by the definitional algorithm or it contains a ground free BG-sorted subterm. If a clause of Cls is included in M^+, then so are all definitions used in that clause. Then that clause is congruent in M^+ to a clause of M in which all finite quantifiers are instantiated. Hence, the congruence closure of M^+ contains a subset of the full instantiation of M, and M is unsatisfiable. □

Example 5.4.1. Let the input formula be $\forall x \in [0,100].\ f(x) > f(0)$ and let M be

$$\begin{array}{lll} \Delta = & x \in [0,100] \setminus \{0\} & \\ (C_1) & f(0) > f(0) & (C_2) \quad f(x) > f(0) \end{array}$$

The modified domain formula $x \in [0,100]$ is the result of one iteration of checkSAT in which find returns $(x, 0)$.
The clause set M^- is

$$\begin{array}{lll} \Delta = & x \in [0,100] \setminus \{0\} & \\ (D_1) & f(0) \approx \alpha_1 & (D_2) \quad f(x) \approx \alpha_2 \\ (C_1) & \alpha_1 > \alpha_1 & (C_2) \quad \alpha_2 < \alpha_1 \end{array}$$

And M^+ is $\{D_1, C_1, C_2\}$, which is unsatisfiable.

Note that each version of M is equivalent, no matter how the clauses are partitioned into FQ-clauses and instances.

5.4.2 Update Heuristic find

This heuristic aims to find a variable x and domain element d such that M^- with $d \in \Delta$ are unsatisfiable, while adding $d \in \Pi$ (but *not* adding the corresponding instance) results in a satisfiable clause set. (Recall the in-place replacement of the constant 1000^i in Section 2). It does this by partitioning domains in M^- to find a maximal satisfiable subset of instances. Then, removing d from Δ_x and simultaneously adding instances formed by the substitution $[x \to d]$ may 'repair' the conjectured equivalence relation on free BG-sorted terms, so that the transformed clause set is satisfiable.

Some of the exception points chosen for refinement can be irrelevant, in the sense that they do not change the clause set beyond adding variants. This problem is addressed with the improved heuristics found in the next chapter.

As usual for such heuristics, time spent searching for a 'good' update pair is traded for the possibility of detecting satisfiability earlier. An advantage of this heuristic over ones presented in the following chapter is that it does not require any modification of the component solver. Its main disadvantage is that much of work done checking subsets of M^- is repeated.

Performance of the algorithm, measured in number of calls to the component solver, scales linearly with the number of domains and at worst logarithmically with the cardinality of the largest domain. That is the reason for working on the clause set as an implication $\Delta \Rightarrow N$ rather than a set of individual FQ-clauses.

```
1  algorithm find((Δ_{x_1} ∧ ... ∧ Δ_{x_n}) ⇒ N)
2  // returns a pair (x_j, d) such that d ∈ Δ_{x_j}
3  for i = 1 to n:
4    let N_i = ⋀ {C ∈ N : FQvars(C) ∩ {x_1,...,x_i} = ∅}
5    if definitional(Δ_{x_{i+1}} ∧ ... ∧ Δ_{x_n} ⇒ N_i) is B-satisfiable:
6      let Γ ⊆ Δ_{x_i} and d ∈ Γ such that
7          definitional(Γ ∧ ... ∧ Δ_{x_n} ⇒ N_i) is B-unsatisfiable and
8          definitional(Γ \ {d} ∧ ... ∧ Δ_{x_n} ⇒ N_i) is B-satisfiable // see text
9      return (x_i, d)
10     // from Lemma 5.4.6 it follows d ∈ Δ_x as claimed
```

Figure 5.3: find determines the next exception point to add

The specific order of domain formulas visited at line 3 is arbitrary, but performance may be improved by sorting by decreasing domain size.

For FQ-clauses with base theory $T_{\mathbb{Z}}$ the set Γ and $d \in \Gamma$ can be determined efficiently, as follows: Assume the set Δ_{x_i} is (a subset of) an interval $[l, u]$ for some numbers l and u with $l < u$. From the above it follows that there is a maximal number u' with $l < u' \leq u$, such that $\Gamma := [l, u'] \cap \Delta_{x_i}$ is as claimed. The number u' can be determined by binary search in the interval $[l + 1, u]$. By maximality, u' is the desired element d. This justifies line 8 in Figure 5.4.2.

Lemma 5.4.6. *Whenever* find *is called from* checkSAT, *then the if-clause on line* 5 *is executed for some* i, *and* find *returns a pair* (x_i, d) *such that* $d \in \Delta_{x_i}$.

Proof. As find is called from checkSAT, it follows that the input set is unsatisfiable. However, the subset with no FQ variables is satisfiable, hence the condition in line 5 in find is satisfied for some i in $1, \ldots, n$. Among all these values, the if-clause is executed for the least one. Specifically, there is some $i \in [1, n]$ for which definitional$(\Delta_{x_i} \wedge \ldots \wedge \Delta_{x_n} \Rightarrow N_i)$ is $\mathcal{B}$-unsatisfiable, while definitional$(\Delta_{x_{i+1}} \wedge \ldots \wedge \Delta_{x_n} \Rightarrow N_{i+1})$ is satisfiable. (If $i = n$, then the satisfiable set is just M^+). Hence, the set of domains $\Gamma \subset \Delta_x$ for which definitional$(\Gamma \wedge \ldots \wedge \Delta_{x_n} \Rightarrow N_i)$ is satisfiable is non-empty, and the maximal such set set is a proper subset of Δ_x. Therefore d exists also. □

5.5 Experimental Results

We have implemented the checkSAT/find algorithm using *Beagle* (Chapter 3) as the component solver. [5] The implementation is prototypical and currently serves only to try out the ideas presented here. Table 5.2 summarizes the experiments performed with this implementation.

[5] This is available in the distribution at http://www.bitbucket.org/peba123/beagle

5.5.1 Problem Selection

Currently, the TPTP problem library does not contain many test problems that exhibit a failure of sufficient completeness. *Beagle* can already solve problems where sufficient completeness and not compactness is at issue, using the Define rule. Hence, test problems are necessarily synthetic. In addition, it is the behaviour of the domain-first refinement algorithm that is under investigation, not the performance of the component solver. This requires synthetic benchmarks in order to minimize the contribution of other factors to the solver's performance, as well as to allow relevant parameters to be tuned.

The problems tested on are listed in Table 5.1. Problems (1) and (6) are $\mathcal{B}$-unsatisfiable, while the remainder are $\mathcal{B}$-satisfiable. Free variables of each problem are quantified over the domain Δ, which is typically of the form $[0, n-1]$ where $|\Delta| = n$; and, for problem (5), $ARRAY(1,2)$ represents the first two axioms of the set $ARRAY$. The only difference between (6) and (6-alt) is the renaming of variables x_2, x_3 to x. This will affect the structure of the finite domains, possibly reducing the difficulty of the problem. For each problem the algorithm was run with a range of domain sizes to better illustrate the scaling behaviour.

#	Status	Statement
1	Unsat	$\forall y.\ f(x) > 1 + y \vee y < 0$
2	Sat	$\forall x.\ x < 0 \Rightarrow g(x) \approx -x \wedge$ $\forall x.\ x \geq 0 \Rightarrow (g(x) \approx x \vee g(x) \approx x + 1) \wedge$ $f(x) < g(x)$
3	Sat	$f(x_1, x_2, x_3, x_4) > x_1 + x_2 + x_3 + x_4$
4	Sat	$f(x) \not\approx x \wedge f(5) \approx 8 \wedge f(8) \approx 5$
5	Sat	$ARRAY(1,2) \wedge \exists a, m.\ (i < j \Rightarrow \mathsf{read}(a,i) \leq \mathsf{read}(a,j) \wedge$ $1 \leq m \wedge m < 1000 \wedge \mathsf{read}(a,m) < \mathsf{read}(a,m+1))$
6	Unsat	$f(x_1) > x_1 \wedge$ $f(x_2 + 3) < 10 \vee \neg x_2 > 2$
6-alt	Unsat	$f(x) > x \wedge$ $f(x + 3) < 10 \vee \neg x > 2$

Table 5.1: Problems used for testing. Free variables range over the domain $\Delta = [0, n-1]$, where the size parameter $n = |\Delta|$ is given in Table 5.2.

In general, the behaviour of the checkSAT algorithm can be understood by dividing problems into categories: Problems can be either satisfiable or unsatisfiable, and dependent or independent of domain size. When satisfiable, the checkSAT algorithm terminates after enough exception points have been added to allow a model. In the unsatisfiable case, the algorithm terminates once an unsatisfiable subset of instances has been found. Problems may also be too difficult for the component solver, this usually results in a timeout on the first call.

Performance on problems, both satisfiable and unsatisfiable, may be independent of the domain sizes. For example, in problem (1) any single instance $[x \to a]$

produces an unsatisfiable set, no matter the size of the domain. Problem (3) is an example of a satisfiable, domain independent problem; it is always satisfiable under the default interpretation. (Notice that the variable y does not need to be finitely quantified). Z3 reports 'unknown' on problem (1), but, surprisingly it solves the similar problem $\mathsf{f}(x) > y \vee y < 0$ quickly. Problems (4) and (5) are also in this category.

Other problems are domain dependent: the size of the smallest set of unsatisfiable ground instances, or the number of exception points required for satisfiability can grow with domain size.

5.5.2 Results

All experiments were carried out on a Linux desktop with a quad-core Intel i7 cpu running at 2.8 GHz, with 8GB of RAM, although the host JVM[6] was configured with maximum heap size of 4GB. Problems were run with a 60 second time limit, executions that exceeded that are marked '-' in the table.

	2			1,3			4		
$\lvert\Delta\rvert$	**#Iter**	**#TP**	**Time**	**#Iter**	**#TP**	**Time**	**#Iter**	**#TP**	**Time**
10	9	40	7.24	1	1	<1	2	7	1.18
20	19	102	13.75	1	1	<1	2	8	1.18
50	-	-	-	1	1	<1	2	10	1.31
100	-	-	-	1	1	<1	2	11	1.32
200	-	-	-	1	1	<1	2	12	1.33
500	-	-	-	1	1	<1	2	13	1.37
1000	-	-	-	1	1	<1	2	14	1.34
2000	-	-	-	1	1	<1	2	15	1.44
5000	-	-	-	1	1	<1	2	17	1.66

	5			6			6alt		
$\lvert\Delta\rvert$	**#Iter**	**#TP**	**Time**	**#Iter**	**#TP**	**Time**	**#Iter**	**#TP**	**Time**
10	3	17	6.18	3	15	2.26	3	12	<1
20	3	19	11.95	3	17	2.28	3	14	<1
50	3	21	19.10	3	19	2.84	3	19	1.1
100	3	23	42.30	3	21	2.18	3	21	1.1
200	-	-	-	3	23	2.26	3	23	1.2
500	-	-	-	3	25	2.47	3	24	1.2
1000	-	-	-	3	27	2.96	3	26	1.3
2000	-	-	-	3	29	3.30	3	28	1.4
5000	-	-	-	3	33	2.95	3	32	1.5

Table 5.2: checkSAT experimental results.

[6]OpenJDK v.1.8

The column '#Iter' is the number of while-loop iterations in checkSAT needed to solve the problem for the given size of Δ. The column '#TP' is the number of theorem prover calls stemming from the various $\mathcal{B}$-satisfiability checks in checkSAT/find. Finally, 'Time' is the total CPU time (seconds) needed to solve the problem.

For comparison, we have also run Microsoft's SMT-solver Z3 [dMB08], version 4.1, on the examples, using the obvious formula representation of the domains Δ.

For problem (2) the function symbol g is 'sufficiently complete' defined by the first two clauses, and only the third clause containing the function symbol f needs finite quantification. Z3 could not solve this problem within three minutes.

Problem (3) was devised to get some insight into Z3's capabilities on the problems described here. While it is trivial for domain-first refinement, Z3 seems to instantiate the clause in problem (3). Clearly, there is a scalability issue here, as for about $|\Delta| > 60$ the problem becomes unsolvable in reasonable time.

As a side note, we found Z3's performance impressive, and it could solve problems (4)–(6) in very short time.

Problem (4) is a simple test of the default interpretation/exception mechanism. Problem (5) is the main example from Section 5.2.

The problems (4) and (6) scale very well, however solving time for (5) increases in linearly with domain size. Note however the difference between (6) and (6-alt): by combining similar domains, run time was halved. This prefigures the enhancement of the next chapter. In problem (4) the exception points are easily discovered from the problem and in (5) the exceptions are quickly discovered by the search. Similarly, in problem (6) the definition for $\mathsf{f}(9)$ is found quickly: the only one needed to establish unsatisfiability. However, this requires searching the domain of x_1 first, then x_2 (comapre Example 5.3.2).

#	$\lvert\Delta\rvert$	*Beagle*	CVC4 Status	CVC4 Time
1	100	<1	Unsatisfiable	0.4
2	10	7.24	GaveUp	0.24
2	20	13.75	GaveUp	0.64
3	200	<1	timeout	
4	200	1.33	Satisfiable	0.9
5	100	42.30	timeout	
6	500	2.47	Unsatisfiable	0.12
6	1000	2.96	Unsatisfiable	0.20
6	2000	3.30	Unsatisfiable	0.28

Table 5.3: checkSAT comparison to CVC4.

Table 5.3 shows the result of running CVC4 (version 1.5)[7] on some representative problems, where the first column (#) references the problem definition in Table 5.1, and the domain size is specified as in Table 5.2 Its recently developed bounded integer finite model finding algorithm builds on [RTGK13], and is rather close in

[7] Using flag –fmf-bounded-int

behaviour to the checkSAT algorithm described here, in the sense that it also builds definitions for uninterpreted terms with variables ranging over finite integer sets. Compared to the default CVC4 configuration, this new feature enables solving the instances of problems 4 and 6.

5.6 Related Work

Procedures for computing models of first-order logic formulas *without background theories* have a long tradition in automated reasoning. MACE-style model finding [CS03] utilizes translation into propositional SAT or into EPR [BFdNT09] for deciding satisfiability w. r. t. a given candidate domain size k; SEM-style model finding [Sla92, ZZ95, McC03] utilizes constraint solving techniques, again w. r. t. k. The main problem is scalability w. r. t. both the domain size k and the number of variables in the input clause set, which severely limits the applicability of both styles in practice. Reynolds et al. [RTG+13, RTGK13] propose a finite model finding procedure in the SMT framework that addresses this problem using on-demand instantiation techniques. However, quantification is restricted to variables ranging into the free sort and the extension to quantification of variables over other interpreted sorts (e. g., integers) is left as future work.

Heuristic instantiation is the state-of-the-art technique for handling quantified formulas in SMT-solvers [GBT07, dMB07]. These heuristics perform impressively well in practice, but are necessarily incomplete. Many language fragments, such as the array property fragment [BMS06] and local theories [IJSS08] admit equisatisfiable translations to finite sets of ground clauses.

Ge and de Moura [GdM09] propose a technique where the ground terms used for instantiation come from solving certain set constraints. They obtain completeness results for the fragment in which every variable occurs only as an argument of a free function or predicate symbol. This fragment is expanded with a subset of LIA terms and also includes fragments such as the array properties fragment. Not all LIA terms are included, for example, terms like $\mathsf{f}(x + y)$ are disallowed, but are acceptable in FQ-clauses when x and y are finitely quantified. Since the procedure is an effective means for *proving* the existence of a finite equivalent set of instances, it can be used to test whether a clause set is eligible to be transformed to a FQ-clause set. This is described in the next section.

The actual quantifier instantiation phase described in Ge and de Moura [GdM09] is known as Model Based Quantifier Instantiation. This involves the SMT solver building a model of the unquantified part of the formula, which is used in an attempt to refute the (negated) quantified formula. If no contradiction is found, then the counter-example model suggests an extension to the existing model. Then, model based quantifier instantiation can be viewed as a way to reduce the set of instantiations which must be explored, and as a way to quickly arrive at a better model for the quantified formula.

Reynolds et al. extend this technique [RTG+13], using the SMT solver to re-

duce the set of instances generated from a given model. Essentially, a given instance of the quantified formula is expanded to a larger set of instances, equivalent modulo the background theory and only instances outside of that set are considered for further rounds of quantifier instantiation. In addition, the model built is constrained to small domain cardinalities by incorporating finite model finding techniques [RTGK13], this then produces a smaller set of terms from which to create formula instances. Although this finite model finding method is refutation complete for uninterpreted first-order quantified formulas, it is not necessarily so for formulas that involve background theories, as it falls victim to the same completeness problems that were described for the Hierarchic Superposition calculus. The finite model finding technique can also be applied to bounded integer domains (FQ-clauses), for similar reasoners as considered here. Uninterpreted functions which range over integers are also given 'definitions' using an expressive constraint language similar to finite domains. However, the focus is on finding a model and relies on theory solvers being capable of providing bindings for shared variables. This is similar to how SMT solvers are employed in the Leon tool described below. As the next chapter shows, using this over and under approximation approach in a refutation theorem proving setting effectively inverts this relationship: checkSAT and find search for *unsatisfiable* sets to refine the definitions. Unsurprisingly then, the SMT based finite model finding approach is model complete (i. e. models are always found for satisfiable clause sets) for FQ-clauses.

Theorem proving using successive under and over-approximation of the problem has been described by Lynch [Lyn04], and is also used to great effect in the Leon static analysis tool for a subset of the Scala language [BKKS13]. While the specific method used by Leon is similar [SKK11a], the domain-first search algorithm differs in that it uses a refutation based first-order theorem prover for discharging proof goals. So the evidence used to find a refinement is a *refutation*, not a model (like in Leon and most SMT based methods), and this necessitates the algorithm find.

Further, the Inst-Gen calculus [GK04a], as used in iProver, can be viewed as employing an approximation strategy, where propositional instantiations are used in the course of a saturation-style proof search. Successive superposition inferences are informed by the result of reasoning on the previous instantiated clause set. An earlier description of the calculus [GK03] also describes a method for obtaining non-ground approximations, useful where those approximations fall in a decidable fragment.

5.6.1 Complete Instantiable Fragments

Where clause sets are in the *array property fragment* or local theories fragment, domain-first search can be applied in lieu of immediate instantiation. The finite set of instances is set as the finite domain which must be searched, possibly using techniques such as those described in Section 5.3.2.

Ge and de Moura [GdM09] describe a way of constructing an equisatisfiable set of instances of a formula using set constraints derived from a clause set. Although they were interested in producing sets of ground instances for consumption by an SMT

solver, the same reasoning can be used to instantiate clauses to the GBT-fragment.

Theorem 5.6.1 (Ge & DeMoura). *Let $\mathcal{N}$ be a clause set and $\Delta_{\mathcal{N}}$ be a set of instances derived from the least solution to set constraints of $\mathcal{N}$. Let $\mathcal{N}^* = \{C\sigma : C \in \mathcal{N}, \sigma \in \Delta_{\mathcal{N}}\}$, then $\mathcal{N} \Leftrightarrow \mathcal{N}^*$.*

An immediate consequence is

Lemma 5.6.1. *Assume $\mathcal{N} \Leftrightarrow \mathcal{N}^*$ as above. Let $\mathcal{N}'$ be any set of instances of $\mathcal{N}$ such that $\mathcal{N}^* \subseteq \mathrm{gnd}(\mathcal{N}')$, then $\mathcal{N} \Leftrightarrow \mathcal{N}'$.*

So a set $\mathcal{N}'$ created by instantiating all variables below BSFG operators is equisatisfiable with the original clause set, as long as those instances are found by solving the set constraints. It is possible that certain clause sets might have a finite set $\mathcal{N}'$ even though the set $\mathcal{N}^*$ is infinite. Such clause sets have equisatisfiable clause sets in the GBT-fragment, and so Hierarchic Superposition is at least refutation complete on those clause sets.

5.7 Summary

This chapter introduced a new method for proving satisfiability of clause sets modulo LIA. The critical restriction is that all integer-sorted, uninterpreted subterms have finitely many instances. This is enforced by bounding certain quantifiers to finite sets. Section 5.3.2 describes how this can be applied to any finite set of instances using an indexing function to integers.

Although the decision problem remains difficult, this technique at least guarantees refutation completeness. And, in some cases, it is more efficient than fully instantiating the clause set, as shown by the experiments.

Inefficiencies still remain: some clauses are already 'complete enough', and do not need to be instantiated with this method. Furthermore, some refinements produced by find can be redundant in the sense that they do not alter the existing clause set under-approximation being tested.

Alternative default interpretation: Taking a constant as the default assignment for free BG-sorted terms is not always a good choice. For example, for the clause $\mathsf{f}(x) \approx x \vee x \notin [1..1000]$ checkSAT needs to amend the default assignment $f(x) \approx a$ at every point in the FQ domain. Any BG-term can be used as a default: in the above, using the term x as the default assignment to $\mathsf{f}(x)$ produces a satisfiable clause set.

Unfortunately the set of terms that can be used in default assignments is limited by the language of the theory solver to be linear polynomials with integer coefficients. For example, while the set $\{\mathsf{f}(x) \approx x \cdot x\}$ has sufficient completeness, the definition is outside of the LIA fragment and so termination of Hierarchic Superposition is no longer guaranteed. Similarly, the use of parameters in a linear polynomial is not possible, e. g. , $\mathsf{f}(x) \approx a \cdot x + b$, for parameters a, b is outside the LIA fragment.

The technique of replacing unknown functions with template terms is described, along with the Model Based Quantifier Instantiation (MBQI) heuristic for SMT solvers [GdM09]. In MBQI, a model for the ground part of the formula is used to interpret terms in the non-ground part. Then the non-ground part is negated and Skolemized, making it eligible for solving with an SMT solver. If that is satisfiable, then the assignments to the Skolem constants in the counter-example are used to produce more ground clauses. Templates are used to replace uninterpreted function symbols in the original problem.

A variant of the above method could be used with checkSAT, using the solution-finding capability described in Chapter 3 for the LIA fragment, or a suitable SMT solver used as a theory solver in the Superposition solver. The critical observation is that clauses in $\mathsf{fixed}(M)$ are equivalent to clauses with ground free BG-sorted terms only. If linear templates are used to replace free BG-sorted terms, then any templates in $\mathsf{fixed}(M)$ will be in the LIA fragment, as they have the form $\alpha_1 d_1 + \ldots + \alpha_k d_k + \beta$ for $d_1, \ldots, d_k \in \mathbb{Z}$. As before, unsatisfiability of $\mathsf{fixed}(M)$ implies overall unsatisfiability, but when satisfiable a binding $v : Consts_{\mathbb{Z}} \mapsto \mathbb{Z}$ can be found for the parameters in the templates.

The binding v is used to rewrite the parameters in M^-, so that all templates are in the LIA fragment. Then one iteration of checkSAT is performed. Again, satisfiability of M^- (modulo v) implies overall satisfiability, otherwise an update (x, d) can found. Adding new clause instances required by the update may expand $\mathsf{fixed}(M)$ and lead to an updated binding v', which can be used in the next iteration of checkSAT. It does not matter that the exception point (x, d) was computed relative to a binding v, as the only effect is to exclude d from the respective finite domain. Termination is not affected either, as once all points are excluded from finite domains, no templates are present and $\mathsf{fixed}(M)$ is equivalent to the original clause set.

In the first iteration $\mathsf{fixed}(M)$ will be empty, so it is necessary to choose an arbitrary non-zero value from the finite domains to instantiate the FQ-clauses.

Bernays-Schönfinkel fragment: The Hierarchic Superposition calculus can immediately be instantiated with, say, an instance-based method for deciding background theories that are given as a set of EPR-clauses. (See Example 2.5.1 in Chapter 2). Then checkSAT, or the extensions above, could possibly be used to integrate arithmetic reasoners, instance-based methods and Superposition. Specifically, in a combination of EPR and arithmetic, any FG operators *must* have a BG result sort, and predicates that take BG terms are disallowed also. So long as every free variable under a BSFG operator is either EPR or finitely bounded, then sufficient completeness can be recovered using the definitional procedure.

Consider applying the checkSAT algorithm to a predicate $p(t)$ which has a finite number of instances, as per the previous example. The default assignment for p will be false. When there are no exception points, the effect will be removing all p-literals from clauses. If the result is satisfiable, then the clause set has a model where p is interpreted as false everywhere. If the result is unsatisfiable, then p can be updated using find. Once an exception point e is found, $p(e)$ is removed

from the default assignment (where it was assigned false) and instances of clauses containing p-literals are added. This continues until either a subset of unsatisfiable clause instances is found or a satisfiable set is found.

In contrast, a model based approach like Model Evolution maintains a *context* that partially specifies the current set of true literals. New literals are added to the context by computing a unifier between the given context and a subset of clauses. When reasoning modulo theories the context unifier may be difficult to compute, even when finite quantification is assumed. The find heuristic in combination with a Hierarchic Superposition solver could offer an alternative.

The next chapter provides a clearer analysis of the reasoning behind the algorithm presented in [BBW14] and Section 5.4. This analysis permits a variety of heuristics for finding the refinement point (alternatives to find), and also a means for applying the update to the model in a way that avoids redundancies. This avoids some pathological cases and points the way to generalizations to other theories, such as that of recursive data structures. Also, some classes of 'complete enough' functions will be described, as alluded to in Examples (1) and (2).

This example motivates some of the modifications in the next chapter:

Example 5.7.1. Consider the following FQ-clauses from some hypothetical Def set, where it is assumed that $\{0,1\} \subseteq \Delta_x$ and $\{0,1\} \subseteq \Delta_y$

$$(1) \quad f(x) \approx \alpha_1 \vee \neg\Delta_x \qquad (2) \quad f(y) \approx \alpha_2 \vee \neg\Delta_y$$

Assume that the first update is $(x, 0)$, so that the new set is

$$(1.1) \quad f(x) \approx \alpha_1 \vee \neg\Delta_x \setminus \{0\} \qquad (2) \quad f(y) \approx \alpha_2 \vee \neg\Delta_y$$
$$(1.2) \quad f(0) \approx \alpha_3$$

Now clauses (1.1) and (2) imply that $\alpha_1 \approx \alpha_2$ as $1 \in \Delta_x \cap \Delta_y$. Moreover, (2) and (1.2) imply $\alpha_2 \approx \alpha_3$, and all together this implies $\alpha_1 \approx \alpha_3$. Because of the presence of variant definitions in Def, no progress was made by the update. In fact, each instance where $x \in \Delta_x \cap \Delta_y$ will need to be added twice as an update. Such a situation can be avoided by careful management of the free BG-sorted terms being defined in definitional, which will be the focus of the next chapter.

Chapter 6

Hierarchic Satisfiability with Definition-First Search

6.1 Motivation

This chapter presents a refinement of the algorithm for hierarchic satisfiability given in the previous chapter. That algorithm was described as a 'domain-first search': default definitions were modified by removing individual points of a finite domain, updating all clauses and definitions in the scope of that particular (bounded) quantifier. In contrast, the new algorithm takes a 'definition-first' approach, in which relevant terms are removed from the default definition with no regard to the domain.

The hierarchic satisfiability procedure works by adding information to the problem to recover completeness (specifically for the Hierarchic Superposition calculus), in a way that prevents excessive generation of clause instances. This is incrementally changed until either satisfiability or unsatisfiability is shown, or all possibilities are exhausted. By giving the definitions a semantic role, i. e. , focusing on what is to be added, the critical *update* step of the procedure can be reasoned about. In particular, the search for the next refinement can be controlled to avoid redundancy (e. g. , where the new under-approximation clause set is logically identical to the previous one).

Recall that the performance of the existing find heuristic also depends on the quantifier structure of the input problem, and has built-in inefficiencies incurred by using the solver in a black-box way. Instead, if the first-order solver provides just the set of definitions used in a proof, then only those need to be searched for an update set. This avoids repeating the same or similar derivations multiple times. Several new heuristics are given that exploit this information, at least one of which does not require finite domains at all.

As domains are no longer explicitly required to organize the definitions and to find updates, the satisfiability algorithm can be used in more general settings where domains are not present, for example, lists and recursive data structures can appear below BSFG operators. Or, the procedure could be used for refutation based solving: unsatisfiability can be deduced by searching for an unsatisfiable set of instances, though the search may not terminate if no such set exists.

This chapter also presents some results on sufficient completeness of *basic definitions* (described in Chapter 4) and the list theory.

Section 6.2 presents the basics of the definition-first algorithm, beginning with a more general version of checkSAT from the previous chapter, then specializing the defining map and approximation steps to the new setting. Section 6.3 introduces the new heuristics enabled by definition-first search. Each different heuristic attempts to minimize some aspect of the unsatisfiable clause set found in the previous approximation step. Section 6.4 gives some experimental results, in particular, comparing the new algorithm with the previous checkSAT algorithm, as well as comparing the various choices of update heuristic. The following sections characterize clauses which can be excluded from the definition-first search: Section 6.5 describes *basic definitions*, similar to those described in Chapter 4; Section 6.6 gives completeness results for data structure theories and gives a version of checkSAT specifically tailored for that case. Finally, Section 6.7 shows that it is theoretically possible to employ a version of checkSAT for refutation complete proof search.

6.2 Definition-First Search

Domain-first search organizes definitions for relevant terms by the domains they share. Exceptions removed individual points from default definitions for all terms in the scope of a selected finite quantifier. However, definitions for relevant terms could also be organized using the term instance relation. For example, take the FQ-clauses

$$\Delta_{x,y} \Rightarrow C[f(x,0)] \wedge D[f(x,y)]$$

with relevant terms being the instances of $f(x,0), f(x,y)$ restricted to the arbitrary finite domain $\Delta_{x,y}$. The domain-first procedure would produce definitions

$$f(x,0) \approx \alpha \vee \neg\Delta_x \qquad f(x,y) \approx \beta \vee \neg\Delta_{x,y}$$

Notice that the definitions assign both parameters α and β to instances of $f(x,0)$. A possible exception instance is $(x,1)$, and this would introduce new definitions

$$f(1,0) \approx \alpha_1 \qquad f(1,y) \approx \beta_1 \vee \neg\Delta_y$$

In definition-first search, each most-general term receives a default definition, and all instances are exceptions to that. In the example the overlap in definitions is corrected, as $f(x,0)$ is an instance of $f(x,y)$:

$$f(x,0) \approx \alpha \vee \neg\Delta_x \qquad f(x,y) \approx \beta \vee \neg\Delta_x \vee \neg\Delta_y \setminus \{0\}$$

The domains are irrelevant, apart from the disequations that enforce the disjointness property of terms and their instances, e.g., $f(x,y) \approx \beta \vee x \not\approx 1 \vee y \not\approx 0$. So definitions could be represented implicitly by a set of terms each being assigned a

unique parameter: non-ground terms assert that all of their instances are equal, *except* for those already contained in the set of defined terms. Exceptions must now be *term instances* rather than variable/integer pairs. Returning to the example, adding an exception $f(1,0)$ updates other domains:

$$f(x,0) \approx \alpha \vee \neg\Delta_x \setminus 1 \qquad f(x,y) \approx \beta \vee \neg\Delta_x \setminus 1 \vee \neg\Delta_y \setminus \{0\}$$
$$f(1,0) \approx \alpha_1$$

Note that domains are retained as the input clauses are FQ-clauses.

As before, the algorithm works on sets of FQ-clauses $\mathcal{N}$, and produces definitions for relevant terms in the input clause set, i.e., for $t \in \mathrm{rel}(\mathcal{N})$. The definition-first search maintains a data structure containing the current set of default definitions organised via the term instance relation[1], called the *defining map* for $\mathcal{N}$. This replaces the domain formula/clause division $\Delta \Rightarrow \mathcal{N}$ of the domain-first search algorithm. As in the previous chapter, definitions assign parameters instead of concrete values, so the BG solver must at least decide satisfiability for the EA-fragment of the BG theory.

This section gives an overview of the new algorithm, then describes defining maps and how they rewrite clause sets, and shows that they meet the requirements for completeness on the over-approximated clause set produced by rewriting with the definitions. Finite domains will still be used to make the relationship with the previous algorithm clear, and because the defining map is simpler when free variables in definitions only range over integer domains. Later it will be shown how defining maps can be used in the general case, in lieu of the syntactic transformation idx described in the previous chapter.

6.2.1 Algorithm

The algorithm presented in Figure 6.2.1 is a modification of $\mathrm{checkSAT}$ from the previous chapter, updated to operate on defining maps. Defining maps are further described in the next section, for now it suffices to consider a defining map for $\mathcal{N}$ $\mathrm{M}_{\mathcal{N}}$ to be a set of ground definitions $t \approx \alpha$ that contains a single definition for each of the relevant terms in $\mathcal{N}$.

The differences between $\mathrm{checkSAT}_{\mathrm{M}}$ and $\mathrm{checkSAT}$ are few, most simply abstract steps of $\mathrm{checkSAT}$ to be independent of the representation of definitions. It will be shown later that the $\mathrm{checkSAT}$ algorithm using defining maps over data structures has roughly the same structure.

As before, the $\mathrm{checkSAT}_{\mathrm{M}}$ procedure takes a set of FQ-clauses $\mathcal{N}$; and maintains a representation of the current set of definitions (here the defining map $\mathrm{M}_{\mathcal{N}}$), which is used to produce over and under-approximations of the input clauses. This representation is updated at the end of each iteration, using information from the previous proof to limit new clause instances introduced. The procedure makes use of the sub-procedures init, apply, $\mathrm{clausal}$, $\mathrm{saturate}$, find and update.

[1] i.e., $t \preceq s$ iff $\exists\sigma.\, t = s\sigma$

```
1  algorithm checkSAT_M(𝒩):
2    let M_𝒩 = init(𝒩)
3    while true:
4      let 𝒩⁻ = apply(M_𝒩, 𝒩) ∪ clausal(M_𝒩)
5      let 𝒩⁺ = { C ∈ apply(fixed(M_𝒩), 𝒩) : C contains no BSFG operators }
6                          ∪ clausal(fixed(M_𝒩))
7      if □ ∈ saturate(𝒩⁺) return UNSAT // Theorem 6.2.4
8      let 𝒟 = saturate(𝒩⁻)  //𝒟 is a saturated clause set
9      if □ ∉ 𝒟 return SAT // Theorem 6.2.3
10     let updateSet = find(M_𝒩, 𝒟)
11     M_𝒩 = update(M_𝒩, updateSet)
```

Figure 6.1: Pseudocode for Definition-First $\mathrm{checkSAT}$ algorithm

The procedure $\mathrm{init}(\mathcal{N})$ constructs an initial defining map for $\mathcal{N}$. There is no constraint on the structure of the initial defining map, so long as it defines each term in $\mathrm{rel}(\mathcal{N})$ (with the possible exception of free BG terms in tautologous clauses). In the implementation init assigns a fresh parameter to each maximal[2] free BG-sorted term of $\mathcal{N}$ up to variants. (Recall a free BG-sorted term is maximal if it is not a proper subterm of another free BG-sorted term).

The procedure apply rewrites the clause set with the current defining map $\mathrm{M}_{\mathcal{N}}$. This should lift the action of rewriting the ground instances of clauses with all combinations of ground instances of definitions in $\mathrm{M}_{\mathcal{N}}$. The resulting clause set should not contain any BSFG operators and should be equisatisfiable with $\mathcal{N}$ modulo the definitions in $\mathrm{M}_{\mathcal{N}}$.

The set $\mathrm{clausal}(\mathrm{M}_{\mathcal{N}})$ is the clausal representation of $\mathrm{M}_{\mathcal{N}}$, which is assumed to have sufficient completeness.[3] As a result, the clause sets $\mathcal{N}^+$ and $\mathcal{N}^-$ are sufficiently complete too. Compared with the previous chapter, apply and $\mathrm{clausal}$ specialize $\mathrm{definitional}$ into two parts which generate sets Cls and Def respectively (where $\mathrm{definitional}$ produced Cls $\cup$ Def). This is because the procedure of rewriting with definitions is more complex when using a defining map. Both apply and $\mathrm{clausal}$ are described in Section 6.2.3.

To form the over-approximation $\mathcal{N}^+$ at line 5, the *fixed* (read: persistent) definitions of the defining map are used. Fixed definitions are those that assign a ground term to a unique parameter, and so the clauses that result from rewriting with a fixed definition appear in all future approximations $\mathcal{N}^-$. This method of over-approximation is independent of the use of finite domains, and can be used in more general applications of $\mathrm{checkSAT}_{\mathrm{M}}$. A definition for fixed is given in the next section. Clauses which contain non-fixed definitions are excluded from $\mathcal{N}^+$, while the fixed definitions are rewritten appropriately by $\mathrm{clausal}$. Both are included in the

[2] In the previous chapter minimal free BG-sorted terms were used, maximal terms are used here to reduce the number of definitions. This will be clarified later.

[3] As in the previous chapter, this requires the inclusion of trivial definitions for terms in $\mathrm{rel}(\mathcal{N})$ that only appear in redundant clauses.

over-approximation $\mathcal{M}^+$.

The procedure saturate calls a Hierarchic Superposition solver. If $\mathrm{saturate}(\mathcal{N}^-)$ terminates and the saturation is consistent, then this indicates that $\mathcal{N}$ is $\mathcal{B}$-satisfiable, as per Theorem 6.2.3. If a contradiction is derived, then $\mathcal{N}$ is unsatisfiable w. r. t. all $\mathcal{B}$-models which also satisfy $\mathrm{M}_{\mathcal{N}}$. As before, $\mathcal{N}^-$ is an under-approximation and must be changed, hopefully in such a way that makes it impossible to repeat the previous derivation from $\mathcal{N}^-$ in the next iteration, while also introducing as few new definitions as possible.

In contrast to the previous version of checkSAT, find operates on the saturated (unsatisfiable) clause set $\mathcal{D}$ (line 9). Information about the proof is encoded along with the clauses, so that only the definitions used in the derivation of unsatisfiability are eligible to be used as exceptions. This clause annotation procedure is described in Section 6.3.1. When a set of exception term instances is found, update inserts those instances as new definitions in the defining map before the next iteration. More information on the find heuristic is in Section 6.3.2.

Theorem 6.2.1. *For FQ-clause set $\mathcal{N}$, $\mathsf{checkSAT}_{\mathrm{M}}(\mathcal{N})$ terminates with the correct result and, if the result is* UNSAT*, then $\mathcal{N}$ with all domain predicates removed is $\mathcal{B}$-unsatisfiable.*

Proof. As the algorithms are similar, the previous proof only needs minor modification. Specifically, now termination is guaranteed by the monotonically increasing set of fixed definitions. New definitions to be fixed are returned by the new version of find on line 10, which always returns a set of terms to fix. Note that if no definitions are returned at line 10, then the derivation of contradiction from $\mathcal{N}^-$ does not depend on any non-fixed definitions made in the defining map M. Specifically, the derivation of unsatisfiability can also be carried out in $\mathcal{N}^+$. Otherwise, the limit of the update process is reached: a defining map with no unsound assumptions, and this provides a definitive check for satisfiability, as shown in Lemma 6.2.4. □

6.2.2 Bounded Defining Map

Defining maps for sets of FQ-clauses will be called *bounded defining maps*, as each of the definitions must be stored with its finite domains. These domains must also be preserved between updates and applied to clauses when rewriting with the definitions.

Definition 6.2.1 (Ground Equivalent)**.** For an FQ-clause $C \vee \neg\Delta$, define the *ground equivalent* $\mathsf{ge}(C \vee \neg\Delta) := \{C\sigma \in \mathsf{vsgi}(C) : \ T_{\mathbb{Z}} \models \Delta\sigma\}$.

The ground equivalent of a set of clauses is the union of the ground equivalents of its members. Because instances that do not satisfy Δ are $T_{\mathbb{Z}}$-valid, it follows that

Lemma 6.2.1. $\mathsf{vsgi}(C \vee \neg\Delta)$ *is equivalent to* $\mathsf{ge}(C \vee \neg\Delta)$ *modulo* $T_{\mathbb{Z}}$.

Sufficient completeness requires every relevant subterm of $\mathcal{N}$ to have a definition. As in the previous chapter, only a subset of the relevant terms will be defined,

namely, the subset of relevant terms in the ground equivalent clause set of $\mathcal{N}$. Sufficient completeness could be recovered immediately by adding arbitrary definitions for relevant terms not in the ground equivalent. It is enough to note that whenever a model exists for the ground equivalent, then there is a model which defines all the relevant terms and is $T_{\mathbb{Z}}$-extending. This is the same as that shown in the previous chapter.

When the input consists of FQ-clauses, each of the free variables in free BG-sorted terms is integer sorted. Definitions introduced as exceptions are formed from non-ground terms in a defining map by instantiating one or more variables with integer values. For example, $f(x) \approx \alpha \vee x \notin [0, 100]$ can have $f(0), f(1)$ added as exceptions. The following restricted form of substitution is used to relate definition terms and instances added as exceptions.

Definition 6.2.2 (Numbering Substitution)**.** A *numbering* substitution is any substitution such that for all $\mathbb{Z}$-sorted variables x, either $x\sigma = x$ or $x\sigma \in \mathbb{Z}$.

A *numbering instance* of a term, literal or clause is any instance made by a numbering substitution.

Example 6.2.1. $[x \to x + 2, y \to 4 + 2], [x \to \alpha], [x \to y]$ are all simple substitutions that are not numbering; $[x \to 6]$ is a numbering substitution.

A defining map for FQ-clauses is represented by a data structure quite similar to a substitution tree used for term indexing, which shares features with the (similarly motivated) defining map used for finite quantifier instantiation in CVC4 [RTG+13], or to a Model Evolution context data structure, described in Baumgartner and Tinelli [BT05]

Definition 6.2.3. A bounded defining map $\mathrm{M}_{\mathcal{N}}$ for a clause set $\mathcal{N}$ is a set of definition and domain formula pairs $(t \approx \alpha, \Delta)$ such that

1. $t \in \mathrm{rel}(\mathcal{N})$
2. α is a $\mathbb{Z}$-sorted parameter not in $\mathcal{N}$
3. all variables in t are in Δ
4. For every maximal term $s \in \mathrm{rel}(\mathsf{ge}(\mathcal{N}))$, there is a definition $(s' \approx \alpha, \Delta) \in \mathrm{M}$ where $s \approx \alpha \in \mathsf{ge}(s' \approx \alpha \vee \neg\Delta)$[4]
5. Given the pairs $(t \approx \alpha, \Delta), (s \approx \beta, \Delta') \in \mathrm{M}_{\mathcal{N}}$ where $\alpha \neq \beta$ then
   (a) t and s are not variants
   (b) if s is a proper numbering instance of t, then $\Delta \cap \Delta' = \emptyset$
   (c) if t, s are not mutual instances or variants, but $\sigma = \mathrm{mgu}(t, s)$ is numbering, then there is $(t' \approx \alpha', \Delta_2) \in \mathrm{M}_{\mathcal{N}}$ where $t\sigma = t'$ and $\Delta_2 \cap \Delta \cap \Delta' = \emptyset$

[4] this will be relaxed later, when terms in $\mathrm{rel}(\mathcal{N})$ are known to be defined by other clauses

Example 6.2.2. Let $\mathcal{N} = \{f(g(a,x),y) \approx y \vee x,y \notin [0,100]\}$. One possible defining map for $\mathcal{N}$ is

$$\begin{array}{llll} f(g(a,x),y) \approx \alpha_0, & x,y \in [1,100] & f(g(a,0),y) \approx \alpha_1, & y \in [1,100] \\ f(g(a,x),0) \approx \alpha_2, & x \in [1,100] & f(g(a,0),0) \approx \alpha_3 & \end{array}$$

The definition $f(g(a,0),0) \approx \alpha_3$ is required by property 5.c of the definition of defining maps. Notice that $g(a,x)$ and $g(a,0)$ are not necessarily defined at this stage. The clausal procedure will do this later, and is described in Section 6.2.3

The following lemma shows that a bounded defining map assigns a single parameter to each relevant term. Note that $\mathsf{ge}(\mathrm{M})$ is the ground equivalent of the set of FQ-clauses $t \approx \alpha \vee \neg\Delta$ where $(t \approx \alpha, \Delta) \in \mathrm{M}$.

Lemma 6.2.2. *Let $\mathcal{N}$ be a set of FQ-clauses and $\mathrm{M}_{\mathcal{N}}$ a bounded defining map for $\mathcal{N}$. For any $t \in \mathrm{rel}(\mathcal{N})$, if $t \approx \alpha_1$ and $t \approx \alpha_2$ are in $\mathsf{ge}(\mathrm{M}_{\mathcal{N}})$, then $\alpha_1 = \alpha_2$.*

Proof. Assume for contradiction that $\alpha_1 \neq \alpha_2$. By property 5a $\mathrm{M}_{\mathcal{N}}$ cannot contain both $t \approx \alpha_1$ and $t \approx \alpha_2$, so these must be instances of two separate definitions. Specifically, $t = s_1\mu_1 = s_2\mu_2$ for substitutions μ_1, μ_2 and $(s_1 \approx \alpha_1, \Delta_1), (s_2 \approx \alpha_2, \Delta_2)$ are in $\mathrm{M}_{\mathcal{N}}$. Hence, s_1, s_2 are unifiable and, without loss of generality, either $t = s_2$ or $t \neq s_2$. If $t = s_2$, then s_2 is an instance of s_1, i. e. , $s_2 = s_1\mu_1$. The substitution μ_1 must be numbering, otherwise $t \approx \alpha_1$ is not in $\mathsf{ge}(\mathrm{M}_{\mathcal{N}})$. But domains Δ_1, Δ_2 are disjoint by property 5b, so μ_1 cannot exist. Finally, if $t \neq s_2$, then by property 5c $t \approx \beta$ is in $\mathrm{M}_{\mathcal{N}}$ and has a domain disjoint from both Δ_1 and Δ_2. □

Now that the defining map has been established as actually being a map, an abstract description of fixed definitions can be given.

Definition 6.2.4 (Fixed definitions)**.** Definition $t \approx \alpha$ is fixed by defining map M if t is ground, and, if any $s \approx \alpha$ is in $\mathsf{ge}(\mathrm{M})$, then $s = t$.

The set of all fixed definitions in M is $\mathsf{fixed}(\mathrm{M})$. For bounded defining maps, each fixed definition has a trivial domain (as it is ground) so the set $\mathsf{fixed}(\mathrm{M})$ consists of unit clauses.

It is possible that a defining map assigns different parameters to terms t, t' such that $T_{\mathbb{Z}} \models t \approx t'$. For example, the defining map $\{(f(x) \approx \alpha_0, x \in [0,5]), (f(x+1) \approx \alpha_1, x \in [0,5])\}$ entails that $f(1) \approx \alpha_0$ and $f(1) \approx \alpha_1$. This particular case could be repaired by transforming the domain predicate $x \in [0,5]$ using the inverse map $x - 1$, followed by combining the definitions. However, the general problem, i. e. , finding common subterms modulo $T_{\mathbb{Z}}$, is essentially theory unification. As requirement 5 in Definition 6.2.3 is only there for sake of theorem prover efficiency, this form of overlap can be ignored without sacrificing completeness. This is not to say the construction of a bounded defining map is in vain, as later experiments will show.

Terms in defining maps may contain instances of other relevant terms as subterms. For example, the subterm $g(a,x)$ in Example 6.2.2 might have a definition $(g(a,x) \approx \beta, \Delta_g)$, but this will not affect any of the other definitions containing that

term. The defining map contains only subterms of the original clause set (i. e. , without any defining map parameters), this has the benefit of keeping the update procedure rather simple.

Bounded defining maps can be viewed as a constraint on the interpretations considered by the solver. As for domain-first search, definition-first search limits instantiation by progressively weakening the constraints imposed by the definitions. Bounded defining maps can be related based on how strict their constraints are: each map organises the relevant terms of a clause set into an equivalence relation, where terms are equivalent if they are assigned the same parameter.

Definition 6.2.5 (Implied Equivalence Relation)**.** A defining map $\mathrm{M}_{\mathcal{N}}$ implies an equivalence $\mathrm{M}^{=}_{\mathcal{N}}$ on a subset of $\mathrm{rel}(\mathcal{N})$, defined as $\mathrm{M}^{=}_{\mathcal{N}} := \{(t_1, t_2) : \exists \alpha.\ t_1 \approx \alpha \in \mathsf{ge}(\mathrm{M}_{\mathcal{N}}) \wedge t_2 \approx \alpha \in \mathsf{ge}(\mathrm{M}_{\mathcal{N}})\}$.

This is a subset of the congruence closure of $\mathsf{ge}(\mathrm{M}_{\mathcal{N}})$. For example, if $f(g(1), 0) \approx \alpha_1, f(g(0), 0) \approx \alpha_2, g(0) \approx \alpha_3, g(1) \approx \alpha_3$ are in $\mathsf{ge}(\mathrm{M}_{\mathcal{N}})$, then $(g(0), g(1)) \in \mathrm{M}^{=}_{\mathcal{N}}$. However, $(f(g(1), 0), f(g(0), 0))$ is not in $\mathrm{M}^{=}_{\mathcal{N}}$, even though the latter terms are equal in the congruence closure of $\mathsf{ge}(\mathrm{M}_{\mathcal{N}})$.

This permits a description of bounded defining maps which abstracts the names of parameters used in the defining map, and provides a way to relate successive defining maps. A defining map is *more general* than another if its implied equivalence relation is a subset of the equivalence relation of the second map. Then the *most general* defining map assigns every relevant term to a unique parameter, equivalently, it only contains ground terms in definitions.

Lemma 6.2.3. *The following are equivalent:*

1. M *is the most general defining map for* $\mathcal{N}$
2. *for all* $t_1 \approx \alpha_1$, $t_2 \approx \alpha_2$ *in* $\mathsf{ge}(\mathrm{M})$, *if* $\alpha_1 = \alpha_2$ *then* $t_1 = t_2$
3. *for all definitions* $(t \approx \alpha, \Delta) \in \mathrm{M}$, t *is ground.*

Proof. 1. $\Rightarrow$ 2. by definition of the implied equivalence on M. 2. $\Rightarrow$ 3. by the fact that all definitions in M are fixed and 3. $\Rightarrow$ 1. by property 5 of Definition 6.2.3. □

Rather trivially then, the most general map does not affect the satisfiability of a clause set, as it uniquely names all existing terms.

Lemma 6.2.4. *Let* M *be the most general defining map for* $\mathcal{N}$*, then* $\mathcal{N} \cup \mathrm{M}$ *is* $\mathcal{B}$*-satisfiable iff* $\mathcal{N}$ *is* $\mathcal{B}$*-satisfiable.*

Lemma 6.2.5. *If* M_1 *is more general than* M_2 *and* $\mathcal{N} \cup \mathrm{M}_2$ *is* $\mathcal{B}$*-satisfiable, then* $\mathcal{N} \cup \mathrm{M}_1$ *is* $\mathcal{B}$*-satisfiable.*

Proof. M_2 differs from M_1 by enforcing additional equalities between relevant terms. Hence, a model I for $\mathcal{N} \cup \mathrm{M}_2$ is already a model for $\mathcal{N} \cup \mathrm{M}_1$ if parameters in definitions are ignored. These can be accommodated by setting $\alpha^I = t^I$, as the terms t in definitions already have interpretations under I. This is not a problem so long as the signature allows the extra constants, as they only appear in definitions of M_1 anyway. □

```
1  algorithm apply(C ∨ ¬ Δ, M_N):
2    let CS = {C ∨ ¬ Δ}
3    while D ∈ CS has maximal relevant term t:
4      for all (s ≈ α, Δ_2) ∈ M_N where mgu(s, t) = σ
5        CS = (CS \ D) ∪ (D[α] ∨ ¬Δ ∨ ¬ Δ_2)σ
6    return CS
```

Figure 6.2: apply rewrites clause $C \vee \neg\Delta$ modulo definitions in $\mathrm{M}_{\mathcal{N}}$

6.2.3 Rewiting Clauses with Defining Maps

In the domain-first search the procedure definitional rewrites the FQ-clause set to the set Cls which is free of BSFG operators, while Def restricts to a subset of possible Σ-interpretations. For the definition-first version, this process is broken into two steps: apply and clausal, which produce Cls and Def respectively.

The procedure apply in Figure 6.2.3 rewrites the input clause set $\mathcal{N}$ with the current defining map. It must preserve the finite domain structure in order to lift the action of rewriting ground clauses with ground definitions.

The procedure apply is similar to definitional in producing the set Cls. It must rewrite maximal terms, since innermost-first rewriting creates new relevant terms which might not appear in the defining map.

Example 6.2.3. The application of the defining map from Example 6.2.2 to

$$f(g(a,1),1) \approx f(g(a,0),y) \vee \neg(y \in [-1,10])$$

gives the clauses

$$\alpha_0 \approx \alpha_1 \vee \neg(y \in [-1,10]) \vee \neg(y \in [-5,20] \setminus 0),$$
$$\alpha_0 \approx \alpha_3$$

Lemma 6.2.6. *If $C'[s_1,\ldots,s_k] \in \mathsf{ge}(C)$ where $s_1,\ldots,s_k$ are all of the maximal free BG sorted terms in C', and $s_i \approx \alpha_i \in \mathsf{ge}(\mathrm{M})$ for each i, then $C'[\alpha_1,\ldots,\alpha_k] \in \mathsf{ge}(\mathrm{apply}(C,\mathrm{M}))$.*

Theorem 6.2.2. *If $\mathsf{apply}(\mathcal{N},\mathrm{M}) \cup \mathrm{M}$ is $\mathcal{B}$-satisfiable then $\mathcal{N}$ is $\mathcal{B}$-satisfiable.*

Proof. Let I be a $\mathcal{B}$-extending model for $\mathsf{apply}(\mathcal{N},\mathrm{M}) \cup \mathrm{M}$. By Lemma 6.2.6 each $C \in \mathsf{ge}(\mathcal{N})$ is in the equational closure of $\mathsf{apply}(\mathcal{N},\mathrm{M}) \cup \mathrm{M}$ hence is entailed by I. By Lemma 6.2.1, $\mathsf{ge}(\mathcal{N})$ is equivalent to $\mathsf{vsgi}(\mathcal{N})$ and so $I \models \mathcal{N}$ because I is $\mathcal{B}$-extending. □

In practice, for a free BG-sorted term t only the most specific generalization of t and all proper instances of t must be included among defining map terms in order to unify with the clause containing t (i.e., the term s at line 4 of apply). This is because, when a definition $t \approx \alpha$ is applied to $C[t']$ and there is a defined term s such that $t\sigma_1 = s$ and $s\sigma_2 = t'$ for non-trivial substitutions σ_1, σ_2, then the disjointness

```
1 algorithm clausal(M):
2   let flat = {(t ≈ α, Δ) ∈ M : t has no proper free BG-sorted subterm }
3   for (t ≈ α, Δ) ∈ M not in flat:
4   let C = t ≈ α ∨ ¬Δ
5   while C = t[s] ≈ α ∨ ¬ (Δ_{x_1} ∧ ... ∧ Δ_{x_k}) has minimal free BG-sorted term s:
6     add s ≈ β ∨ ¬ (⋀_{x ∈ vars(s)} Δ_x) to flat
7     let C = t[β] ≈ α ∨ ¬ (⋀_{x ∈ vars(t[β])} Δ_x)
8   add C to flat
9 return flat
```

Figure 6.3: clausal transforms defining map M to a clause set without affecting sufficient completeness

constraint (in the domain formula) of $t \approx \alpha$ will be falsified, making the resulting clause redundant.

Next, clausal in Figure 6.2.3 transforms definitions to FQ-clauses, the most important step of which is flattening.

Note that a definition equation $(t \approx \alpha, \Delta_t)$ from a bounded defining map is equivalent to an FQ-clause: $t \approx \alpha \vee \neg\Delta_t$ this is guaranteed by points 1 and 2 of the bounded defining map definition. Lemma 6.2.2 prevents the immediate derivation of $\alpha_i \approx \alpha_j$ from a such a clausal representation. However, this direct translation to clauses may produce a clause set that does not have sufficient completeness.

The behaviour and structure of non-ground defining maps is greatly simplified if definitions are kept only for maximal free BG-sorted terms. For example, $f(g(a,x),y) \approx \alpha$ is used, rather than something of the form $f(z,y) \approx \alpha, g(a,x) \approx \beta$. However, the former definition will not have sufficient completeness if $g(a,x)$ is not covered by any definition. This is further illustrated in the following example:

Example 6.2.4. Consider the set of ground definitions, where f, g are BSFG operators

$$\{f(g(\mathbf{n})) \approx \alpha : \forall \mathbf{n} \in \mathbb{N}\} \cup$$
$$\{g(f(\mathbf{n})) \approx \beta : \forall \mathbf{n} \in \mathbb{N}\}$$

This cannot cover $\mathrm{rel}(\mathcal{N})$ for a clause set $\mathcal{N}$, since it must also include definitions for each $f(\mathbf{n})$ and $g(\mathbf{n})$. There is a model for Δ that is not a $T_{\mathbb{Z}}$-model:

Let I be an interpretation that assigns the carrier set $\mathbb{Z} \cup \{\omega\}$ to the sort $\mathbb{Z}$, where ω is some non-integer element. Define f^I, g^I such that for all $i \in \mathbb{Z}$, $f^I(i) = g^I(i) = \omega$ and let $f^I(\omega) = g^I(\omega) = 0$. Then, the given clause set does not have sufficient completeness.

The replacement of s in t by a constant yields an equivalent clause only when s is ground. Otherwise, it introduces the assumption that instances of s are equivalent. This does not change the result of apply, as the terms affected by the new assumption are already identified under the original (maximal) definition. For example, $f(g(a,x),y) \approx \alpha$ is flattened to $f(\beta,y) \approx \alpha$, $g(a,x) \approx \beta$, meaning terms such

as $f(g(a,0),1)$ and $f(g(a,1),1)$ are made equal. However, those terms are already equal due to the definition $f(g(a,x),y) \approx \alpha$. As with other assumptions in the defining map, the new assumption can be repaired by the update procedure described later. That modification is accomplished by changing the definition for the maximal term $t[s] \approx \alpha$, rather than creating a separate definition for s.

Lemma 6.2.7. *If I is a $\mathcal{B}$-extending interpretation such that $I \models \mathsf{clausal}(\mathrm{M})$, then $I \models \{t \approx \alpha \vee \Delta : (t \approx \alpha, \Delta) \in \mathrm{M}\}$*

Proof. $\mathsf{ge}(M)$ is contained in the equational closure of $\mathsf{ge}(\mathsf{clausal}(\mathrm{M}))$ by construction. The simplification step at line 6 of the procedure does not remove any clauses from the ground equivalent set. □

The following theorem shows that $\mathcal{N}^-$, as defined in $\mathrm{checkSAT}_M$, is indeed an under-approximation of $\mathcal{N}$.

Theorem 6.2.3. *Given a set of FQ-clauses $\mathcal{N}$, if $\mathcal{N}^- = \mathsf{apply}(\mathrm{M}_\mathcal{N}, \mathcal{N}) \cup \mathsf{clausal}(\mathrm{M}_\mathcal{N})$ is satisfiable, then $\mathcal{N}$ is $\mathcal{B}$-satisfiable.*

Proof. As for Theorem 4.4 in the last chapter, any model of $\mathcal{N}^-$ can be made into a $\mathcal{B}$-extending model. Then by Theorem 6.2.2 and Lemma 6.2.7 it follows that $\mathcal{N}$ is $\mathcal{B}$-satisfiable. □

The set $\mathsf{fixed}(\mathrm{M})$ contains definitions and rewritten clauses which do not change in any more-general defining maps. Again, for bounded defining maps, the set $\mathcal{N}^+$ is simply the subset of definitions and clauses without finite quantified variables after rewriting with definitions.

Lemma 6.2.8. *If M_1 and M_2 are both defining maps for $\mathcal{N}$, and M_1 is more general than M_2, then $\mathsf{fixed}(\mathrm{M}_2) \subseteq \mathsf{fixed}(\mathrm{M}_1)$.*

This immediately implies the following

Theorem 6.2.4. *If $\mathcal{N} \cup \mathsf{fixed}(\mathrm{M})$ is $\mathcal{B}$-unsatisfiable, then $\mathcal{N}$ is $\mathcal{B}$-unsatisfiable.*

Proof. By Lemma 6.2.8, $\mathcal{N}^+$ is a subset of $\mathsf{apply}(\mathrm{M}_{max}, \mathcal{N}) \cup \mathsf{clausal}(\mathrm{M}_{max})$, where M_{max} is the most general defining map for $\mathcal{N}$. By Lemma 6.2.4, it follows that $\mathcal{N}$ is $\mathcal{B}$-unsatisfiable. □

6.3 Updating Defining Maps

Progress in the domain-first search algorithm is made using the find heuristic to identify a finite domain and a value to remove from that domain, such that the subset of clauses and definitions with that value excluded was satisfiable. The same method could be used to make progress in definition-first search, but defining maps provide a way to compute more accurate updates more efficiently than find. This requires a modification of the component solver and an abstract characterization of what makes a 'good' update. The current section will formalize 'updates', describe

```
1  algorithm update((t ≈ α, Δ_t), M):
2    let Δ_0 = Δ_t
3    for (s ≈ β, Δ_s) ∈ M:
4      if σ = mgu(r, s) is numbering:
5        if s is ground:
6          remove σ from Δ_0
7        else:
8          replace (s ≈ β, Δ_s ∧ ¬Δ_t) in M
9    for r ≈ α ∈ ge(t ≈ α ∨ ¬ Δ_0):
10     let α_r be a fresh parameter
11     add (r ≈ α_r, ∅) to M
```

Figure 6.4: Procedure for applying a single update $(t \approx \alpha, \Delta_t)$ to defining map M.

the clause labelling scheme used to find the subset of clauses and definitions to pass to find, and finally, give three alternative implementations for find with different performance characteristics.

An *update* for a defining map is a subset of definitions from the current defining map that is unsatisfiable when taken together with the input clause set . Of course, the set of all definitions is unsatisfiable (assuming that $\mathrm{satisfiable}(\mathcal{N}^-)$ fails), but large update sets introduce more instances on the next iteration of $\mathrm{checkSAT}$. Therefore, any update heuristic should aim to minimize the number of terms in the update set, while preserving unsatisfiability of $\mathcal{N}^-$. In particular, if an update set is minimal (in that any subset does not produce an unsatisfiable set), then all of its terms should be updated in the defining map, not just a single one.

Definition 6.3.1 (Update)**.** An update for a defining map is a set U of ground terms $\{t \in \mathrm{rel}(\mathcal{N}) : t \text{ is not fixed in } \mathrm{M}\}$.

The action of the procedure update shown in Figure 6.3 is to replace the existing definitions for update terms with equations between update terms and fresh BG parameters. Update sets involving multiple definitions can be applied using multiple calls to update.

It simply removes shared instances in existing definitions (line 8) to enforce the disjointness property of defining maps (property 5 in Definition 6.2.3), then adds all ground instances of $(t \approx \alpha)$ not already present with a fresh parameter replacing α. The removal at line 6 prevents duplicate additions.

After the application of an update, all terms in the update are fixed by the new defining map, by the simple fact that all instances of the updated term are present as ground terms in the new defining map.

Lemma 6.3.1. *Let* M' *be the defining map produced by* $\mathsf{update}((t \approx \alpha, \Delta), \mathrm{M})$*, for some term* t *and defining map* M*. Then for every* $s \in \mathsf{ge}(t \approx \alpha \vee \neg\Delta)$*, it follows that* $s \in \mathsf{fixed}(\mathrm{M}')$.

6.3.1 Clause Labels

Now that definitions are 'atomic', their use can be traced through a proof to show which clauses depend on which specific definitions in the current defining map. Clause labels will be used for this: A *clause label* is an extra-logical set of definitions stored with clauses, that contains the non-ground definitions used to rewrite a clause in the apply procedure. Labels are meant to represent the use of an unsound assumption in the derivation, which is eligible for inclusion in an update set. A labelled clause is written $C \mid L$, where L is a possibly empty set of labels. Here labels are simply definition/domain pairs, e. g. , $(t \approx \alpha, \Delta)$.

For example, applying a non-fixed definition $(f(x,y) \approx \alpha, \Delta)$ to $C[f(z,1)] \vee \neg\Delta_z$ produces labelled clause $C[\alpha] \mid (f(z,1) \approx \alpha, \Delta \wedge \Delta_z)$ (braces in label sets are typically omitted for clarity). The definition in the clause label has been modified with the unifier $[x \to z, y \to 1]$ used in applying the definition to the clause.

The procedures apply, $\mathrm{clausal}$ defined above are modified to include labels as follows:

- In apply each clause initially receives an empty label, and whenever a definition D rewrites a clause $C \mid L$ with substitution σ the label L is extended with $D\sigma$.
- In $\mathrm{clausal}$ each definition to be flattened is labelled by itself, and labels are preserved through flattening.

Clause labels are passed through inferences and simplifications: the conclusion takes the union of labels of the premises along with any unifier (or matcher) used in the inference (or simplification). The existing calculus rules are modified in the following general way[5]:

$$\frac{C_1 \mid L_1 \qquad C_2 \mid L_2}{C_3\sigma \mid (L_1 \cup L_2)\sigma_{num}}$$

where σ is the inference substitution and σ_{num} is its restriction to only numbering substitutions, or renamings on the variables of $L_1 \cup L_2$ (i. e. , bijections on variables).

Then every clause in a derivation from labelled clauses will be labelled with the definitions necessary to derive it. Since only non-fixed definitions are used in labels, these roughly correspond to unsound assumptions used in the proof. Specifically, these assumptions are that all instances of defined terms are equal. The application of unifiers to labels means that a clause label may identify only a subset of defined term instances which are necessarily assumed to be equal.

The following example shows how different defining maps may block or allow certain derivations.

Example 6.3.1 (Preventing inferences with definitions)**.** Fix some finite domain $\Delta_{x,y}$, and let $\mathrm{M} = \{(f(x,y) \approx \alpha, \Delta_{x,y})\}$. This defining map applies to a single clause

[5]Except for the optional non-deterministic split rule, which is disabled for this application as it can separate clauses from their domain formulas.

$x \not\approx f(x,y) \vee g(x,y) \not\approx g(0,0) \vee \neg\Delta_{x,y}$ (assuming g does not have a BG result sort), producing the labelled clause

$$x \not\approx \alpha \vee g(x,y) \not\approx g(0,0) \vee \neg\Delta_{x,y} \mid (f(x,y) \approx \alpha, \Delta_{x,y})$$

Consider an application of equality resolution

$$\frac{x \not\approx \alpha \vee g(x,y) \not\approx g(0,0) \vee \neg\Delta_{x,y} \mid (f(x,y) \approx \alpha, \Delta_{x,y})}{0 \not\approx \alpha \vee \neg(\Delta_{x,y}[x \to 0, y \to 0]) \mid (f(0,0) \approx \alpha, \emptyset)}[x \to 0, y \to 0]$$

Notice that the unifier $[x \to 0, y \to 0]$ applies to the label, making its domain formula ground. Equivalently, it represents an empty set. If $x \approx 0$ and $y \approx 0$ are not excluded a priori by $\Delta_{x,y}$, the domain part of the conclusion $\neg(\Delta_{x,y}[x \to 0, y \to 0])$ simplifies to $\neg \mathit{true}$, and it can be removed.

Next, let $f(0,0)$ be fixed by taking $\mathrm{M}' = \{(f(x,y) \approx \alpha, \Delta_{x,y} \wedge x \not\approx 0 \wedge y \not\approx 0), (f(0,0) \approx \alpha', \emptyset)\}$. The new labelled clause is

$$x \not\approx \alpha \vee g(x,y) \not\approx g(0,0) \vee \neg(\Delta_{x,y} \wedge x \not\approx 0 \wedge y \not\approx 0) \mid (f(x,y) \approx \alpha, \Delta_{x,y} \wedge x \not\approx 0 \wedge y \not\approx 0)$$

The only difference is that the finite domain is now restricted. Similarly, in the inference above the finite domain is replaced everywhere with the new restricted version, and, as a result, the conclusion becomes trivially true.

Hence, one way to prevent the derivation of a labelled clause is to add one of the definitions of the labels as a new definition. In general, if (d, Δ) is a definition, then adding instance $(d[x \to n], \Delta[x \to n])$ to the defining map will also require modifying the original definition to $(d, \Delta \wedge x \not\approx n)$[6]. This creates a tautological clause if the substitution $[x \to n]$ is applied to a clause with the modified domain. However, clause domains can be modified without the change being recorded in the label. For example, in $(C[x \to \alpha] \mid L)$ assuming x is a finitely quantified variable, then the substitution $[x \to \alpha]$ does not apply to the label L, as α is a parameter. If $(C[x \to \alpha] \mid L)$ is demodulated with $\alpha \approx 5$, say, then only the definition instances in $L[x \to 5]$ are really necessary. But that information is lost due to the parameter substitution.

Though such cases could probably be accounted for, for simplicity *all* instances of a definition in a label will be fixed in the new defining map. This is guaranteed to prevent the derivation of the given labelled clause, as at least one ancestor clause is completely removed by instantiation.

A successful proof produces either a labelled empty clause or a $\mathcal{B}$-unsatisfiable set of labelled Σ_{B}-clauses. Given a derivation of an empty clause, then any instantiation of the set of clauses used to derive it will also produce an empty clause; yet another reason to add *all* instances of a label definition.

In summary, adding all instances of at least one definition in the label of the empty clause in a derivation from $\mathcal{N}^-$ should block the same proof being derived

[6]Compare this with the final step of checkSAT in the previous chapter.

in the next iteration. However, derivations that end with an empty clause and no invocation of the BG solver are the rare easy case. The more common case of an unsatisfiable set of labelled BG clauses is the focus of the next section. It will be necessary to select some set of definition instances such that at least one definition in each clause label in the unsatisfiable set is completely covered.

6.3.2 Finding Update Sets

It is desirable to extract a small but relevant update from a derivation of an unsatisfiable set of labelled BG clauses. 'Small' refers to the number of terms in the ground equivalent of the update set. A small update will generate only a few new clauses in the next iteration of $\mathrm{checkSAT}_{\mathrm{M}}$. 'Relevant' means that it should not allow repeating the current derivation under the updated defining map. This is already guaranteed by the use of clause labels.

As described in the previous section, the derivation of a $\mathcal{B}$-unsatisfiable clause set can be blocked with an update that includes all instances of at least one definition from every label of a clause in that set. There are two ways to minimize the size of this update: either minimize the number of labels selected for the update, or minimize the size of the $\mathcal{B}$-unsatisfiable clause set while preserving unsatisfiability.

A *minimal hitting set* for labels is a set of definitions which contains a definition from each clause label and is minimal w. r. t. sets with that property. A *minimal unsatisfiable core* (MUC) of a $\mathcal{B}$-unsatisfiable clause set is a $\mathcal{B}$-unsatisfiable subset of which any proper subset is $\mathcal{B}$-satisfiable.

The following heuristics combine those minimizations in slightly different ways, with different aggregate behaviour. As the update process is a trade-off between time used searching for a small update versus the time lost by including unnecessary instances in the defining map, there is no definite choice of one over the other.

Ground MUC: Some SMT solvers are capable of efficiently finding a MUC from among large sets of ground formulas. The $\mathcal{B}$-unsatisfiable clause set can be instantiated to ground clauses using the domain formulas of the labels. Quantifier elimination (Cooper's algorithm) can be used to remove any variables that do not appear in labels.

Lemma 6.3.2. *A $\mathcal{B}$-unsatisfiable set of labelled Σ_{B}-clauses derived from a set of FQ-clauses can always be ground instantiated.*

Proof. The algorithm is largely as follows:

- Use background theory quantifier elimination to eliminate free variables in $C \mid L$ that do not occur in L. The result is a labelled formula $\phi \mid L$.
- For each labelled clause or formula add all instances over the label domains

The result is a conjunction of labelled ground instances. □

```
1 algorithm NG-MUC(𝒟):
2   for label l in labelSet(𝒟):
3     𝒟₁ = 𝒟 with all clauses labelled by l removed
4     if 𝒟₁ is sat:
5       core = core ∪ {l}
6     else:
7       𝒟 = 𝒟₁
8   return core
```

Figure 6.5: Pseudocode for the NG-MUC heuristic

The SMT solver returns a MUC of ground clauses, the labels of which are searched for a minimal hitting set of (ground) definition terms to use as an update.

The disadvantage of this approach is that the number of ground clauses produced for testing still scales exponentially with the size of domains. On the other hand, SMT solvers are optimized for larger problem instances than first-order solvers, and generally the final $\mathcal{B}$-unsatisfiable clause set is smaller than the input clause set.

Non-Grounding MUC: This approach minimizes the number of label definitions selected, without instantiating the clause set. Optionally, once a set of label definitions is selected, sets of clause instances that do not contribute to unsatisfiability are removed by bisecting the domains of the selected definitions. Since this can only be done for FQ-clauses, the combination of non-grounding MUC search and domain bisection is a separate heuristic (described later).

The following algorithm minimizes the set of labels by removing any clauses from the $\mathcal{B}$-unsatisfiable set of Σ_B-clauses $\mathcal{D}$ that are labelled by a selected label definition, then continuing with a different selected label until the set becomes satisfiable. The BG solver is not required to be able to find a MUC, as the method is implemented as an outer-loop around an existing BG solver. Also, the NG-MUC procedure does not depend on the representation of the labels or the presence of domain formulas, so the heuristic could be used for more general formula fragments.

For example take the following set of clauses labelled with definitions d_1, d_2, d_3

$$(1)\ \ 0 < \alpha \mid d_1, d_2 \qquad (2)\ \ \alpha < 1 \mid d_2 \qquad (3)\ \ \alpha \not\approx 0 \mid d_3$$

Clearly $0 < \alpha$ and $\alpha < 1$ together are unsatisfiable. The result of a run of NG-MUC is $\{d_1, d_2\}$, the set of labels of that minimal unsatisfiable core. At this point, whichever definition in $\{d_1, d_2\}$ has the least number of instances could be returned, since the removal of any definition in $\{d_1, d_2\}$ gives a $\mathcal{B}$-satisfiable clause set (e. g. , removing all clauses labelled with d_2 yields just clause (3), which is satisfiable). This is also the main disadvantage of NG-MUC: each of the label definitions after minimization may contain many new instances. For example, both d_1 and d_2 could be $(g(x) \approx \alpha_g, x \in [0, 100])$. This can happen when no unifier used in the derivation of the unsatisfiable clause set acts on any finitely quantified variable. There is no

```
1  algorithm reduce(𝒟, Labels):
2   for l ∈ Labels:
3    for x ∈ vars(l) where x ∈ [m, n] ⊂ ℤ:
4     do:
5      𝒟_r = replace each C | l[x], ... ∈ 𝒟
6            with C ∨ ¬(m ≤ x ∧ x ≤ (m + (n − m)/2))
7      n = m + (n − m)/2
8     while (𝒟_r is unsat & m ≠ n)
9     update bounds for x in l
10  return Labels // with updated bounds
```

Figure 6.6: The reduce heuristic builds on NG-MUC by subdividing domains

immediately apparent way to reduce the number of those instances without using the finite domains somehow, as described for the next heuristic.

Domain Reduction: The set of instances of those definitions in core, as returned by NG-MUC, is minimized by reducing the size of the FQ-domains of variables in the selected literal set (the argument Labels). This heuristic operates on the assumption that the finite domains in the clause part are at least a subset of the finite domains in the label. Finite domains in clauses are reduced by adding extra literals, e. g. , to reduce $x \in [0, 100]$ by half, the literal $x < 50$ would be added to the clause. Rather than attempt to compute exactly which elements remain in the finite domain, this is approximated by taking the minimum and maximum values from the label domain m, n respectively, on line 3 of the code listing.

As for NG-MUC this reduction procedure is repeated for each variable in each label as long as the clause set remains unsatisfiable.

This step could potentially involve many calls to the BG solver, so it should be implemented with a timeout. Also, the procedure could stop as soon as a single ground label is produced (i. e. , the domains of all of its free variables have been reduced to single elements), and use that as the update.

Comparable applications are found in: Torlak et al. [TCJ08] which translates a declarative specification into a SAT problem then searches for a minimal unsatisfiable core in the specification; also in Ryvchin and Strichman [RS11] which looks for 'high-level' unsatisfiable cores, considering clauses to be grouped into sets which are added or removed as a whole. Both applications use at base a simple minimization algorithm for finding a core: it removes clauses one at a time, testing satisfiability, until any subset of the remaining clauses is satisfiable, just as in NG-MUC.

6.4 Experimental Results

We have produced a prototype implementation to investigate the scaling behaviour of the heuristics given above and to show any improvement over the original, modular

#	$\lvert\Delta\rvert$	Status	Statement
1	100	Unsat	$\forall y.\ f(x) > 1 + y \vee y < 0$
2	10	Sat	$\forall x.\ x < 0 \Rightarrow g(x) \approx -x \wedge$ $\forall x.\ x \geq 0 \Rightarrow (g(x) \approx x \vee g(x) \approx x + 1) \wedge$ $f(x) < g(x)$
3	200	Sat	$f(x_1, x_2, x_3, x_4) > x_1 + x_2 + x_3 + x_4$
4	200	Sat	$f(x) \not\approx x \wedge f(5) \approx 8 \wedge f(8) \approx 5$
5	1000	Sat	$\mathrm{ARRAY}(1,2) \wedge \exists a, m.\ (i < j \Rightarrow \mathsf{read}(a,i) \leq \mathsf{read}(a,j) \wedge$ $1 \leq m \wedge m < 1000 \wedge \mathsf{read}(a,m) < \mathsf{read}(a,m+1))$
6	500	Unsat	$f(x_1) > x_1 \wedge$ $f(x_2 + 3) < 10 \vee \neg x_2 > 2$

Table 6.1: Same problems as in Chapter 5, Table 5.1 but with fixed domain cardinality

#	Original	Z3-MUC	NG-MUC	NG+red
1	1.39	1.36	5.33	1.66
2	9.71	4.23	3.43	4.11
3	1.59	1.45	1.60	1.62
4	1.93	1.40	1.61	1.63
5	(timeout)	2.48	2.73	2.81
6	4.26	3.42	2.65	5.51

Table 6.2: Run time in seconds of four solver configurations on the problems.

algorithm. Both use the *Beagle* solver [7] to implement the $\mathrm{saturate}$ call. As in [BBW14], we are not looking to evaluate either the pure first-order or the pure background reasoning performance of the solvers (each being handled by a modular sub-solver), but rather how performance scales with respect to the size of the finite quantification domains. The original problems from [BBW14] remain illustrative of the categories of behaviour that $\mathrm{checkSAT_M}$ may exhibit, and they allow a comparison of the performance of the old and new versions.

As seen in Chapter 3, most of the problems from the TPTP library which extend $T_{\mathbb{Z}}$ were already solved by *Beagle* , and the few that were not solved failed for reasons other than those addressed here (e. g. , problems with non-linear multiplication or compactness), hence an analysis of performance on those problems is not relevant.

In Table 6.1, the free variables of each problem are quantified over the domain Δ, which is typically of the form $[0, n-1]$ where $|\Delta| = n$; and for problem five, $\mathrm{ARRAY}(1,2)$ represents the first two axioms of the set ARRAY.

In Table 6.2 the columns are, from left to right: the original $\mathrm{checkSAT}$ algorithm, as described in Chapter 5 and [BBW14]; minimal unsatisfiable core with instantiation, using the SMT solver Z3 [dMB08] to find the core; non-ground minimal unsatisfiable core, corresponding to the $\mathrm{NG\text{-}MUC}$ algorithm described above; minimal unsatisfi-

[7] http://bitbucket.org/peba123/beagle

$\lvert\Delta\rvert$	**Z3-MUC**	**NG-MUC**	**NG+red**
10	3.10	2.10	4.60
50	15.12	8.91	12.72
100	25.14	12.52	18.51
150	37.39	15.10	34.70
200	(timeout)	20.95	39.10

Table 6.3: Scaling behaviour (run time in seconds) on problem two.

able core with domain reduction, which is the sequential execution of both NG-MUC and reduce.

NG-MUC performs worse on problem one because that problem requires only a single instance to yield unsatisfiability, whereas NG-MUC selects a non-ground definition, producing a large update set. Each of the other methods are able to return a singleton update set. All new versions perform significantly better on problems two and five due to the duplication in reasoning over variant terms that is implicit in the original checkSAT algorithm. (If $f(x)$ and $f(y)$ appear in separate clauses, then they are given separate definitions in the original algorithm, with up to $|\Delta_x|$ extra exceptions that may be added).

In Table 6.3, checkSAT_M solves problem two by giving a value for the function g at each point in the domain. Then the Z3-MUC method must instantiate clauses over the entire domain on each iteration of the satisfiability algorithm, while only adding a small update set. The result is that many large clause sets are tested for little benefit in terms of proof progress. In contrast, both of the non-grounding algorithms avoid full instantiation up until the final iteration, at which point satisfiability is shown.

Interestingly, the domain reduction does not improve overall performance, due to the symmetry of the problem. It is only when all values of g are added to the defining map (in any order) that the problem is shown satisfiable. In other problems, especially unsatisfiable ones (such as problem one) NG+red has an advantage, as it can narrow down on the necessary instances quickly and the overhead of the extra BG solver calls pays off.

Critically, it appears that the efforts to reduce duplication of reasoning have produced improvements over the original method, showing that its advantages are retained in the more general account given here. At least one update heuristic NG-MUC is independent of both a requirement for finding a MUC and for having FQ-clauses specifically.

6.5 Sufficient Completeness of Basic Definitions

This section introduces *basic definitions*, simple syntactic patterns which guarantee sufficient completeness (of a subset of the clauses), and which can easily be identified.

This is critical in conjunction with checkSAT procedures (both in this and the last chapter), because relevant terms which are already defined by clauses do not

need to be given default definitions in a defining map. Consequently, the number of refinement steps in a run of checkSAT may be drastically reduced if those definitions are excluded.

Definition 6.5.1 (Basic definition)**.** A *basic definition* is a unit clause

$$f(s_1, \ldots, s_k) \approx t$$

where $f : S_1 \times \ldots \times S_k \to \mathrm{B}$; (B is the BG sort) t is a Σ_B-term and $\mathrm{vars}(t) \subseteq \mathrm{vars}(f(s_1, \ldots, s_k))$.

Recall the basic flat definitions from Chapter 4, these required each of $s_1, \ldots, s_k$ to be variables. As the name implies, they are also basic definitions. Allowing terms instead of variables is acceptable when proving sufficient completeness, but requires an extra check, given in Lemma 6.5.1.
Consider the axioms (1) from LIST$[E]$ and (2) from ARRAY$[E]$:

$$\mathsf{head}(\mathsf{cons}(x, y)) \approx x \qquad \mathsf{read}(\mathsf{store}(v, w, x), w) \approx x$$

Each of these is a basic definition. A clause set including these definitions has local sufficient completeness if any instance of $\mathsf{head}(\mathsf{cons}(x, y))$ or $\mathsf{read}(\mathsf{store}(v, w, x), w)$ in the clause set have only Σ_B-terms in the x variable position. Otherwise, the set of relevant terms includes terms of the form, e. g. , $\mathsf{head}(\mathsf{cons}(t, y))$, which rewrite to free BG-sorted terms t, and not to pure BG terms, as required in the sufficient completeness definition.

Lemma 6.5.1. *Given a clause set $\mathcal{N}$ such that for each term $r \in \mathrm{rel}(\mathcal{N})$ there is a basic definition $f(s_1, \ldots, s_k) \approx t \in \mathcal{N}$ and substitution σ, where $f(s_1, \ldots, s_k)\sigma = r$ and $t\sigma$ is a Σ_B-term. Then $\mathcal{N}$ has local sufficient completeness.*

Proof. Let $\mathcal{M} \models \mathrm{sgi}(\mathcal{N}) \cup \mathrm{GndTh}(B)$ and take r, t as above. By assumption, there is a substitution σ such that $t\sigma$ is a Σ_B-term, and since $\mathrm{vars}(t) \subseteq \mathrm{vars}(f(s_1, \ldots, s_k))$ it follows from the fact that $f(s_1, \ldots, s_k)\sigma$ is ground, that $t\sigma$ is ground. Therefore $\mathcal{N}$ has local sufficient completeness. □

In addition, defining maps can combine with basic definitions without destroying sufficient completeness. Leaving basic definitions untouched by the satisfiability procedure reduces the number of definitions to be updated, greatly improving the efficiency of the definition-first search.

First, a lemma about combining clause sets with sufficient completeness.

Lemma 6.5.2. *If $\mathcal{N}_1, \mathcal{N}_2$ are Σ-clause sets such that both have local sufficient completeness, then $\mathcal{N}_1 \cup \mathcal{N}_2$ has local sufficient completeness.*

Proof. The result follows from $\mathrm{rel}(\mathcal{N}_1 \cup \mathcal{N}_2) = \mathrm{rel}(\mathcal{N}_1) \cup \mathrm{rel}(\mathcal{N}_2)$. Very-simple substitutions cannot make a term into a free BG-sorted term, so terms in $\mathrm{rel}(\mathcal{N}_1 \cup \mathcal{N}_2)$ are very-simple instances of terms in either $\mathcal{N}_1$ or $\mathcal{N}_2$. □

Theorem 6.5.1. *Given a FQ-clause set $\mathcal{N}$ and defining map* M *for $\mathcal{N}$ such that every $t \in \mathsf{ge}(\mathcal{N})$ is either a simple instance of the left-hand side of a basic definition in $\mathcal{N}$; or, for some parameter α, $t \approx \alpha$ is in* $\mathsf{ge}(\mathrm{M})$*. Then if* $\mathsf{clausal}(\mathrm{M}) \cup \mathsf{apply}(\mathrm{M}, \mathcal{N})$ *is satisfiable, it is $\mathcal{B}$-satisfiable.*

Proof. Let $\mathcal{N} = \mathcal{D} \cup \mathcal{N}'$ where $\mathcal{D}$ is the set of all basic definitions in $\mathcal{N}$. Note that $\mathsf{vsgi}(\mathcal{D}) \cup \mathrm{M}$ fulfils the conditions for a defining map for $\mathcal{N}$ (apart from possibly number 5). Let $\mathcal{K} = \{t \approx \alpha : t \in \mathrm{rel}(\mathcal{N})$ but is not a subterm of a clause in $\mathsf{ge}(\mathcal{N})\}$, where α is the same for each term t. Since LHS terms of equations in $\mathcal{K}$ only occur in trivial ground instances of $\mathsf{vsgi}(\mathcal{N})$, it follows that $\mathcal{K}$ does not affect the satisfiability of the set, in particular $\mathcal{N} \cup \mathrm{M} \cup \mathcal{K}$ implies $\mathcal{N} \cup \mathrm{M}$. By Lemma 6.5.2, $\mathsf{clausal}(\mathrm{M}) \cup \mathsf{apply}(\mathrm{M}, \mathcal{N}') \cup \mathcal{D} \cup \mathcal{K}$ has local sufficient completeness, and following the proof of Theorem 4.2 in the previous chapter, $\mathsf{clausal}(\mathrm{M}) \cup \mathsf{apply}(\mathrm{M}, \mathcal{N}') \cup \mathcal{D}$ is $\mathcal{B}$-satisfiable. □

6.6 Sufficient Completeness of Recursive Data Structure Theories

The data structure theories given in Chapter 2 are 'almost' sufficiently complete, in that the axioms define the interaction of the FG elements (lists, arrays, trees etc) with the BG theory well enough that most relevant terms satisfy the condition for sufficient completeness. This section gives proofs of sufficient completeness, or lack thereof, of those data structure theories with integer element theory (so that selectors are BSFG operators). The proofs suggest a use of the definition-first procedure to recover completeness for extensions of those theories, in which data structres are defined by templates in an iterative deepening style.

For an example of the use of basic definitions, consider the theory ARRAY with $\mathbb{Z}$ as the index and element sort. Recall that $\Sigma_{\mathrm{ARRAY}} = \{\mathsf{read} : ARRAY \times \mathbb{Z} \to \mathbb{Z}, \mathsf{store} : ARRAY \times \mathbb{Z} \times \mathbb{Z} \to ARRAY, a_0 : ARRAY\}$ and ARRAY is:

$$(1)\quad i \approx j \vee \mathsf{read}(\mathsf{store}(m,i,e),j) \approx \mathsf{read}(m,j) \qquad (2)\quad \mathsf{read}(\mathsf{store}(m,i,e),i) \approx e$$
$$(3)\quad (\forall i.\ \mathsf{read}(m,i) \approx \mathsf{read}(n,i)) \Rightarrow m \approx n \qquad (4)\quad \mathsf{read}(a_0,i) \approx 0$$

Note that axiom (4) is new and defines the constant array a_0 so that the set of very-simple ground instances of a term with free array sorted variables is well-defined. As mentioned above, axioms (1) and (4) of ARRAY are basic definitions, and therefore do not need to be included in the defining map.

The axiom set ARRAY does not have sufficient completeness as, for example:

$$\mathrm{ARRAY}[\mathbb{Z}] \models_{\mathbb{Z}} \mathsf{store}(a_0,0,0) \approx a_0,$$

but $\mathsf{vsgi}(\mathrm{ARRAY}[\mathbb{Z}]) \not\models \mathsf{store}(a_0,0,0) \approx a_0$

Lemma 6.6.1. *The axiom set* $\mathrm{ARRAY}[\mathbb{Z}]$ *does not have sufficient completeness.*

Proof. The transformation of (3) to CNF produces a new Skolem function $sk : \mathcal{A} \times \mathcal{A} \to \mathbb{Z}$. Roughly, sk is expected to return an index at which the argument arrays differ, or some arbitrary index if they are the same. Then $sk(a_1, a_2) \in \mathrm{rel}(\mathrm{ARRAY}[\mathbb{Z}])$ for array terms a_1, a_2. Assume that some $(\Sigma_{\mathbb{Z}} \cup \Sigma_{\mathrm{ARRAY}})$-interpretation $\mathcal{M}$ assigns a non-integer value ω to $sk(a_1, a_2)$. Then $\mathcal{M}$ is free to assign any value to $\mathsf{read}(a_1, \omega)$ and $\mathsf{read}(a_2, \omega)$ since no clause in $\mathrm{vsgi}(\mathrm{ARRAY}[\mathbb{Z}])$ apart from the instantiation of (3) with a_1, a_2 contains (a term interpreted as) the value ω. It is then possible to have $\mathcal{M} \models \mathsf{read}(a_1, i) \approx \mathsf{read}(a_2, i)$ for every $i \in \mathbb{Z}$, but also have $\mathcal{M}(\mathsf{read}(a_1, \omega)) \neq \mathcal{M}(\mathsf{read}(a_2, \omega))$ so that $a_1 \approx a_2$ is not entailed by $\mathcal{M}$. □

Let $\Sigma_{\mathrm{LIST}[\mathbb{Z}]} = \{head : LIST \to \mathbb{Z}, tail : LIST \to LIST, cons : \mathbb{Z} \times LIST \to LIST, nil : LIST\}$ and $\mathrm{LIST}[\mathbb{Z}]$:

$$
\begin{array}{ll}
(1) \quad \mathsf{head}(\mathsf{cons}(x, l)) \approx x & (2) \quad \mathsf{tail}(\mathsf{cons}(x, l)) \approx l \\
(3) \quad l \approx \mathsf{nil} \vee \mathsf{cons}(\mathsf{head}(l), \mathsf{tail}(l)) \approx l & (4) \quad \mathsf{nil} \not\approx \mathsf{cons}(x, l) \\
(5) \quad \exists x.\, \mathsf{head}(\mathsf{nil}) \approx x & (6) \quad \mathsf{tail}(\mathsf{nil}) \approx \mathsf{nil}
\end{array}
$$

Note the extra axioms (5) and (6), which allow deducing $\mathsf{head}(\mathsf{tail}^n(\mathsf{nil})) \approx \mathsf{head}(\mathsf{nil})$ for $n \geq 0$.

Lemma 6.6.2 ([AKW09])**.** *The axiom set* $\mathrm{LIST}[E]$ *has sufficient completeness for any element theory* E*.*

Proof. Terms in $\mathrm{rel}(\mathrm{LIST}[E])$ have the form $\mathsf{head}(l)$ where l is either nil, $\mathsf{cons}(e, l')$, or $\mathsf{tail}(l')$ where e is a Σ_{B}-term. The first two cases are covered by axioms (1) and (5), which are basic definitions. For the last case, note that terms of the form $\mathsf{tail}(l')$ can be reduced to $\mathsf{cons}(l'')$ or nil by axioms (2) and (6), so $\mathsf{head}(\mathsf{tail}(l'))$ can be proven equal to some Σ_{B}-term too. □

Clause sets extending the list theory lose sufficient completeness as soon as list constants other than nil are introduced. For example, the clause set $\mathrm{LIST}[\mathbb{Z}] \cup \{\mathsf{head}(l) \not\approx x\}$ where l is a list constant, has a model[8] in which $\mathsf{head}(l)$ is interpreted as an arbitrary non-integer element.

Armando et al. [ABRS09] show that with a specific ordering and transformation of the clause set the superposition calculus finitely saturates the set $\mathrm{LIST} \cup \mathcal{G}$ where $\mathcal{G}$ is a set of quantifier free literals (unit clauses). The crucial point is that literals in $\mathcal{G}$ are flattened and reduced by LIST. Then the only relevant terms that occur are ground or are subterms of clauses in LIST– in basic definitions in fact. Put differently: by restricting to quantifier-free literals, unbounded access in arbitrary lists cannot be defined. In that case Hierarchic Superposition with the Define rule is complete (assuming the correct term order).

Theorem 6.6.1. *Given a ground conjunction* $\mathcal{G}$ *of flat* $(\Sigma_{\mathrm{LIST}[\mathbb{Z}]} \cup \Sigma_{\mathbb{Z}})$*-literals where the* $\Sigma_{\mathbb{Z}}$ *part is in a complete fragment [BW13a], then, with an appropriate ordering, Hierarchic Superposition with weak abstraction decides* $T_{\mathbb{Z}} \cup \mathrm{LIST}[\mathbb{Z}]$*-satisfiability of* $\mathrm{LIST}[\mathbb{Z}] \cup \mathcal{G}$*.*

[8] more specifically $\mathrm{sgi}(\mathrm{LIST}[\mathbb{Z}] \cup \{\mathsf{head}(l) \not\approx x\}) \cup \mathrm{GndTh}(T_{\mathbb{Z}})$ has a model

Proof. As per the reasoning in Armando et al. [ABRS09] Superposition inferences produce a finite saturated clause set, and the ordering constraints on the Hierarchic Superposition calculus (specifically, that domain elements are minimal) do not contradict the ordering required for saturation of LIST axioms. As Hierarchic Superposition inferences are a subset of possible (unsorted, non-theory) Superposition inferences considered by [ABRS09], it follows that the clause set is saturated w. r. t. Hierarchic Superposition also. The saturated clause set without the LIST axioms contains only ground free BG-sorted terms, and by [BW13b], Hierarchic Superposition with weak abstraction is (refutation) complete on that fragment. Together with Lemma 6.6.2, if the result does not contain the empty clause, then $T_{\mathbb{Z}} \cup \text{LIST}[\mathbb{Z}]$-satisfiability follows. □

By generalizing Lemma 6.6.2, more general clauses can be supported. This motivates the satisfiability procedure for lists described in Section 6.6.1.

Theorem 6.6.2. *Let $\mathcal{N}$ be a Σ-clause set where $\Sigma_{\text{LIST}[E]} \subseteq \Sigma$ and whose only BSFG operator is* head. *If for every* LIST*-sorted subterm l of clauses in* $\mathsf{vsgi}(\mathcal{N})$, $\mathcal{N} \cup \text{LIST} \models l \approx t$ *for some term t consisting only of* cons, nil *and ground Σ_{B}-terms, then $\mathcal{N} \cup$ LIST has sufficient completeness.*

As a practical application of Theorem 6.6.2, adding definitions for list operators such as map, sum and length as per Chapter 4 should not affect sufficient completeness, so long as the definitions are well-founded. However, those definitions often assume a well-founded order on model elements (such that $\mathsf{cons}(l) \succ l$ for any l say). This assumption is violated in the presence of infinite or cyclic lists. In decision procedures, such as in Oppen [Opp80], satisfiability is w. r. t. the theory of acyclic lists, but in this case where the list theory is described axiomatically, infinite lists are not excluded. By including the length operator which maps lists into an interpreted theory, interpretations of Σ_{LIST} including infinite lists can be excluded by the $T_{\mathbb{Z}}$-extending criterion for models of saturated clause sets.

Define $\mathsf{length} : \mathit{list} \to \mathbb{Z}$ by

$$(1) \quad \mathsf{length}(\mathsf{cons}(x, y)) \approx \mathsf{length}(y) \qquad (2) \quad x \not\approx \mathsf{nil} \vee \mathsf{length}(x) \approx 0$$

Theorem 6.6.3. LIST$[E]$ *with* length *operator has sufficient completeness and any $T_{\mathbb{Z}}$-extending model does not have any infinite length lists.*

Proof. Clause sets over $\Sigma_{\text{LIST}[E]} \cup \{\mathsf{length}\}$ have sufficient completeness, by a similar argument to 6.6.2. List constants are allowed, since terms $\mathsf{length}(\mathsf{tail}^n(l))$ reduce to $\alpha - n$ using the Define rule. Let I be a $T_{\mathbb{Z}}$-extending model of the axioms LIST$[E]$ and clauses 1) and 2) of the length definition. Then for any list element w, $I(\mathsf{length}(w)) = k \in \mathbb{Z}$, so $I \models \mathsf{length}(\mathsf{tail}^k(w)) \approx 0$ and $\mathsf{tail}^k(w) \approx \mathsf{nil}$ by 2). So no $w \in I$ is infinite. □

6.6.1 Recursive Data Structure Definitions

This application of definition-first search exploits the fact that only the individual list constants must be defined in order for LIST$[E]$ to have sufficient completeness.

[9] This tacitly assumes there are no other non-constant LIST sorted operators in the signature other than cons.

A *list defining map* for a clause set must include a definition for each list sorted constant appearing in the clauses, except for nil. Given a clause set $\mathcal{N}$ with subterms including Σ_{LIST} operators, let $\mathcal{L}$ be the set of LIST-sorted constants other than nil in $\mathcal{N}$. There are three types of list defining maps for $\mathcal{N}$:

$$\begin{aligned}
\mathrm{M}^{\emptyset}_{\text{LIST}} &:= \{l \approx \mathsf{nil} \,:\, l \in \mathcal{L}\} \\
\mathrm{M}^{-}_{\text{LIST}} &:= \{l \approx \mathsf{cons}(e_{l1}, \ldots, \mathsf{cons}(e_{lk}, \mathsf{nil}) \ldots) \,: \\
&\qquad \text{for fresh constants } e_{li},\ k \geq 1 \text{ and } l \in \mathcal{L}\} \\
\mathrm{M}^{+}_{\text{LIST}} &:= \{l \approx \mathsf{cons}(e_{l1}, \ldots, \mathsf{cons}(e_{lk}, l_{\text{end}}) \ldots) \,: \\
&\qquad \text{for fresh constants } e_{li}, l_{\text{end}} \text{ where } k \geq 1 \text{ and } l \in \mathcal{L}\}
\end{aligned}$$

The defining map $\mathrm{M}^{+}_{\text{LIST}}$ effectively fixes a minimum length for each list constant, while $\mathrm{M}^{-}_{\text{LIST}}$ assigns an exact length individually, for each list constant. Hence, maps of the form $\mathrm{M}^{+}_{\text{LIST}}$ are called *indefinite*, while those of form $\mathrm{M}^{-}_{\text{LIST}}$ (i. e. , each list term terminated by a nil constant) are *definite*.

Example 6.6.1. Let $\mathcal{L} = \{l_1, l_2, l_3\}$.

$$\begin{aligned}
\mathrm{M}^{\emptyset}_{\text{LIST}} &= \{l_1 \approx \mathsf{nil}, l_2 \approx \mathsf{nil}, l_3 \approx \mathsf{nil}\} \\
\mathrm{M}^{-}_{\text{LIST}} &= \{l_1 \approx \mathsf{nil}, l_2 \approx \mathsf{cons}(e_1, \mathsf{cons}(e_2, \mathsf{nil})), l_3 \approx \mathsf{cons}(e_4, \mathsf{nil})\} \\
\mathrm{M}^{+}_{\text{LIST}} &= \{l_1 \approx l_{1,\text{end}}, l_2 \approx \mathsf{cons}(e_1, \mathsf{cons}(e_2, l_{2,\text{end}}), l_3 \approx \mathsf{cons}(e_4, l_{3,\text{end}}\}
\end{aligned}$$

Since a list defining map does not contain any BSFG operators at all, it does not need to be flattened or otherwise transformed to ensure sufficient completeness. Also, applying the defining map to a clause set is simply a matter of replacing list constants with their new definitions.

Clause sets extended with either $\mathrm{M}^{\emptyset}_{\text{LIST}}$ or $\mathrm{M}^{-}_{\text{LIST}}$ have sufficient completeness by Theorem 6.6.2. As with Section 2, $\mathrm{M}^{-}_{\text{LIST}}$ is an under-approximation: satisfiability can be deduced with this restriction, but not unsatisfiability. A clause set extended with list defining map $\mathrm{M}^{+}_{\text{LIST}}$ does not have sufficient completeness, however Theorem 6.6.4 shows that it can be used to over-approximate a clause set, similar to how the set of fixed definitions is used above.

Define a relation $\preceq$ on list defining maps over a set $\mathcal{L}$, where $\mathrm{M}_1 \preceq \mathrm{M}_2$ if for all $l \in \mathcal{L}$ the depth of the terms assigned to l by M_1 is less or equal to the depth of the term assigned by M_2. Then from the previous example $\mathrm{M}^{\emptyset}_{\text{LIST}} \preceq \mathrm{M}^{-}_{\text{LIST}}$ and $\mathrm{M}^{-}_{\text{LIST}} \preceq \mathrm{M}^{+}_{\text{LIST}}$.

Theorem 6.6.4. *Let $\mathrm{M}^{+}_{\text{LIST}}$ be an indefinite list defining map for clause set $\mathcal{N}$. If, for all definite defining maps $\mathrm{M}'_{\text{LIST}} \preceq \mathrm{M}^{+}_{\text{LIST}}$, it is the case that $\mathcal{N} \cup \mathrm{M}'_{\text{LIST}}$ is unsatisfiable, and $\mathcal{N} \cup \mathrm{M}^{+}_{\text{LIST}}$ is unsatisfiable, then $\mathcal{N}$ is unsatisfiable.*

[9] It is possible, though inefficient, to transform clause sets including LIST$[T_{\mathbb{Z}}]$ to FQ-clauses using the idx transform in Chapter 5

```
1  algorithm checkSAT_LIST(𝒩):
2    let M = M^∅_LIST
3    let updateQueue = []
4    while (true):
5      let 𝒟 = saturate(𝒩 ∪ M_LIST)
6      if (□ ∉ 𝒟) return SAT
7      listConsts = find(𝒟)
8      if (listConsts.isEmpty) return UNSAT
9      if (□ ∈ saturate(𝒩 ∪ M^−LIST)) return UNSAT
10      updateQueue.push(listConsts)
11     M = extendList(updateQueue.pop())
```

Figure 6.7: $\mathcal{N}$ is a set of clauses including LIST$[\mathbb{Z}]$

Proof. Assume that $\mathcal{N} \cup \mathrm{M}^{+}_{\mathrm{LIST}}$ is unsatisfiable. Then for all list constants l, where $l \approx \mathsf{cons}(e_{l1}, \ldots, \mathsf{cons}(e_{lk}, l_{\mathsf{end}}) \ldots) \in \mathrm{M}^{+}_{\mathrm{LIST}}$, there are no models of $\mathcal{N}$ in which l has at least k elements. By hypothesis, there are no models where l has fewer than k elements, therefore $\mathcal{N}$ has no models. □

Unlike the corresponding proof for bounded defining maps, this proof relies on the fact that prefixes of the hypothesised lists have already been shown unsatisfiable. So the refininement procedure can lengthen just one list by one element each iteration. The choice of which list constant will take the modified definition is still open, and the same clause labelling procedure can be used to restrict the choice to just those that label a minimal unsatisfiable set of BG-clauses.

A version of the $\mathrm{checkSAT}_{\mathrm{M}}$ satisfiability algorithm specialized to the LIST theory is given in Figure 7.

The procedure $\mathrm{find}(\mathcal{D})$ checks the labelled clauses in the saturated clause set $\mathcal{D}$ and returns a set of list constants for which the assumption 'has length exactly k' is required for the proof. If there are no such constants, then the derivation is independent of $\mathrm{M}^{-}_{\mathrm{LIST}}$, and so the same derivation is possible using $\mathrm{M}^{+}_{\mathrm{LIST}}$. Therefore, it is correct to conclude UNSAT. Otherwise, those constants are pushed onto the global update queue.

At line 9 the defining map $\mathrm{M}^{+}_{\mathrm{LIST}}$ is a list defining map identical to $\mathrm{M}^{-}_{\mathrm{LIST}}$, except with all nil constants replaced with fresh list constants. In addition, the call to $\mathrm{saturate}$ on line 8. could have a timeout imposed, as Theorem 6.6.4 only requires definite maps to be checked.

So long as calls to $\mathrm{saturate}$ are terminating, the procedure $\mathrm{checkSAT}_{\mathrm{LIST}}$ will produce a counter-example, where one exists. This is not possible simply using the $\mathrm{saturate}$ procedure alone, as there is no way to determine whether the implied model is a $T_{\mathbb{Z}}$-model. Since no assumptions are made on the structure of $\mathcal{N}$, termination is not guaranteed a priori, and, in general, inductive facts cannot be proven either.

The $\mathrm{checkSAT}_{\mathrm{LIST}}$ procedure can also be modified to search over other recursive data structures, e. g. , binary trees, given a suitable modification of Theorem 6.6.4 and

a corresponding fair enumeration of possible shapes for the given data structure.

6.7 Refutation Search

Now that definitions can be represented independently of their finite domains, it is possible to discard the finite domains entirely. For similar reasons as above, each individual test of an approximation clause set can yield a correct $\mathcal{B}$-satisfiable or $\mathcal{B}$-unsatisfiable result.

This section will sketch a method for refutation search based on the $\mathrm{checkSAT}_{gen}$ algorithm above, showing that with an appropriate implementation of find and a global fairness criterion it is possible to obtain refutation completeness. A few open questions remain, and are described at the end of the section. In particular, the given restrictions on heuristics are likely to be inefficient in practice, however more efficient heuristics may be feasible, when restricted to fragments for which satisfiability is equivalent to a finite set of clause instances.

In this section, like in other sections, all substitutions will be simple. The defining map used here is similar to a bounded defining map, except with no finite domain structure. Instead, *dismatching constraints* will be used to ensure that instances of definitions do not overlap.

Definition 6.7.1 (Dismatching Constraint, [GK04b])**.** A dismatching constraint is a pair of term tuples $\mathsf{ds}(\bar{s}, \bar{t})$, such that $\bar{s}$ and $\bar{t}$ are variable disjoint. A substitution σ is a member of $\mathsf{ds}(\bar{s}, \bar{t})$, when for all (simple) substitutions γ, $\bar{s}\gamma = \bar{t}\sigma$.

The critical point is that a dismatching constraint is falsified exactly when there is a matcher μ such that $\bar{s}\mu = \bar{t}$.

Dismatching constraints can be joined via conjunctions, this corresponds to intersection of the sets of member substitutions. It is assumed that in conjunctions $\wedge_{i=1}^{n} \mathsf{ds}(\overline{s_i}, \overline{t_i})$, each s_i is variable disjoint with t_j and s_k for $k \neq i$.

All equations in defining maps will be constrained with dismatching constraints; a non-ground definition without a constraint represents the set of *all* of its simple ground instances. These are constructed so that instances of definitions do not overlap (see point (2) in the following definition). Example 6.7.1 will show how to construct such dismatching constraints.

Definition 6.7.2 (General Defining Map)**.** Given a set of terms T, a *general defining map* for T is a set of equations with constraints $\mathrm{M}_T = \{s_1 \approx \alpha_1 \mid D_1, \ldots, s_n \approx \alpha_n \mid D_n\}$, where parameters α_i are fresh; each s_i is an instance of some term $s'_i \in T$ (that has a BSFG operator outermost) and each D_i is a dismatching constraint conjunction. M_T must have the following properties:

1. Given $s_1 \approx \alpha_1 \in M_T$, there is no $s_2 \approx \alpha_2 \in \mathrm{M}_T$ such that $\mathrm{mgu}(s_1, s_2) = \sigma$ is renaming and non-empty. If it is empty, then $\alpha_1 = \alpha_2$.

2. If $s_1 \approx \alpha_1 \mid D_1$ and $s_2 \approx \alpha_2 \mid D_2$ are in M_T where $\mathrm{mgu}(s_1, s_2) = \sigma$ is not a renaming substitution, then $s_1\sigma \approx \beta \mid D_3$ is in M_T for some parameter β,

and there are no ground substitutions γ_i, γ_j in D_i, D_j such that $s_i\gamma_i = s_j\gamma_j$ for distinct i, j in $\{1, 2, 3\}$.

3. For every $s \in T$ there is a definition $s' \approx \alpha \in \mathrm{M}_T$ for some parameter α such that there exists matcher μ where $s'\mu = s$.

Usually the set of terms T is the set of relevant terms (i. e. , very-simple ground instances of free BG-sorted terms) for a clause set.

To generate a dismatching constraint for $t \approx \alpha \in \mathrm{M}_T$, take the set Inst_t of substitutions μ such that

1. $t\mu \approx \beta \in \mathrm{M}_T$ and
2. there is no $t' \approx \beta' \in \mathrm{M}_T$ where $\exists\gamma, \gamma'.\ t\gamma = t'$ and $t'\gamma' = t\mu$

From each matcher $[x_1 \to t_1, \ldots, x_m \to t_m]$ in Inst_t create the dismatching constraint $\mathsf{ds}(\langle t_1, \ldots, t_m\rangle, \langle x_1, \ldots, x_m\rangle)$, then $t \approx \alpha$ is constrained with the conjunction of all such constraints generated from the set.

Example 6.7.1 (Dismatching Constraints)**.** A defining map for term set

$$T = \{f(x,y), f(g(z),y), f(g(a),y), f(x,b)\} \text{ is:}$$

$$\begin{aligned}
f(x,y) &\approx \alpha_0 \mid \mathsf{ds}(b,y) \wedge \mathsf{ds}(g(z),x) & \qquad f(g(a),y) &\approx \alpha_2 \mid \mathsf{ds}(b,y)\\
f(g(z),y) &\approx \alpha_1 \mid \mathsf{ds}(a,z) \wedge \mathsf{ds}(b,y) & \qquad f(g(z),b) &\approx \alpha_4 \mid \mathsf{ds}(a,z)\\
f(x,b) &\approx \alpha_3 \mid \mathsf{ds}(g(z),x) & \qquad f(g(a),b) &\approx \alpha_5
\end{aligned}$$

Brackets from single element tuples are left out for clarity. For example, $[x \to a, y \to a]$ is a member of the constraint for $f(x,y) \approx \alpha_0$, but $[x \to a, y \to b]$ and $[x \to g(a), y \to b]$ are not.

The first step is to show that the clause set approximations $\mathcal{N}^+$ and $\mathcal{N}^-$ work as intended when using a general defining map.

$\mathcal{N}^+$ is the simplest of the two approximations and consists of just the ground (fixed) definitions in the defining map, along with all clauses (instances) in $\mathrm{vsgi}(\mathcal{N})$ which are completely rewritten by those fixed definitions.

Lemma 6.7.1. *$\mathcal{N}^+$ unsatisfiable implies $\mathcal{N}$ is unsatisfiable.*

Proof. Since $\mathcal{N}^+$ is equivalent to a subset of $\mathrm{vsgi}(\mathcal{N})$. □

'Ground equivalent' in this context means all very-simple ground instances that satisfy any dismatching constraints present.

Definition 6.7.3 (Ground Equivalent for Dismatching Constraints)**.** Given a constrained clause $C \mid D$, then $\mathsf{ge}(c \mid D)$ is the set $\{C\sigma : \sigma \in D \text{ and } C\sigma \text{ is ground}\}$.

The apply procedure is the same as before, only dismatching constraints are preserved instead of finite domains. Dismatching constraints are also used in the derivation to block inferences that would trivialize the defining map.

Example 6.7.2 (Constraint Simplification Required)**.** The unit clause $f(x', b) \approx t[x']$ is rewritten into three clauses by the defining map in Example 6.7.1:

$$\begin{aligned} \alpha_3 &\approx t[x] \mid \alpha_3 \cdot \mathsf{ds}(g(z), x) \\ \alpha_4 &\approx t[g(z)] \mid \alpha_4 \cdot \mathsf{ds}(a, z) \\ \alpha_5 &\approx t[g(a)] \mid \alpha_5 \end{aligned}$$

The parameter in the constraint denotes the unique definition equation with that parameter. If constraints are ignored, there is an obvious superposition inference between the second and third clauses, producing $\alpha_4 \approx \alpha_5$ with unifier $[z \to a]$. This violates the dismatching constraint $\mathsf{ds}(a, z)$ and should be removed. This same pattern will occur each time definitions in which the defined terms are proper instances (e. g. , $f(g(z), b) \approx \alpha_4$ and $f(g(a), b) \approx \alpha_5$) rewrite the same clause.

Again, this is necessary to make progress in the $\mathrm{checkSAT}_{gen}$ loop.

Note also, that due to the lack of finite domains, there is no need to complete $\mathcal{N}^-$ to a sufficiently complete clause set, as it has sufficient completeness already.

Lemma 6.7.2. $\mathsf{ge}(\mathsf{apply}(\mathcal{N}, \mathrm{M}_{\mathrm{rel}(\mathcal{N})}) \cup \mathsf{flat}(\mathrm{M}_{\mathrm{rel}(\mathcal{N})}))$ *has sufficient completeness.*

Proof. As a result of Definition 6.7.2, $\mathrm{M}_{\mathrm{rel}(\mathcal{N})}$ has a definition for all relevant terms in $\mathcal{N}$. So $\mathsf{apply}(\mathcal{N}, \mathrm{M}_{\mathrm{rel}(\mathcal{N})})$ does not contain any free BG terms. Any term in $\mathrm{vsgi}(\mathsf{flat}(\mathrm{M}_{\mathrm{rel}(\mathcal{N})}))$ is equated to some parameter, and so is necessarily equal to some BG element in any model of $\mathsf{ge}(\mathsf{apply}(\mathcal{N}, \mathrm{M}_{\mathrm{rel}(\mathcal{N})}) \cup \mathsf{flat}(\mathrm{M}_{\mathrm{rel}(\mathcal{N})})) \cup \mathrm{GndTh}(\mathcal{B})$ □

Lemma 6.7.3. *If $\mathcal{N}^-$ is produced by rewriting with $\mathrm{M}_{\mathrm{rel}(\mathcal{N})}$ and is satisfiable, then $\mathcal{N}$ is $\mathcal{B}$-satisfiable.*

Proof. Note that simplification of dismatching constraints never removes any clauses from the ground equivalent. By Lemma 6.7.2, if $\mathcal{N}^-$ is saturated w. r. t. HSP, then it is $\mathcal{B}$-satisfiable. By construction it has a $\mathcal{B}$-extending model, which also satisfies $\mathrm{M}_{\mathrm{rel}(\mathcal{N})}$. As $\mathcal{N}$ is implied by the equational closure of $\mathcal{N}^-$, it follows that the given model is also a model of $\mathcal{N}$. □

These two lemmas establish that the 'local' behaviour of $\mathsf{checkSAT}_{gen}$ is correct, however the global behaviour is still unspecified– does it ever terminate? Is termination guaranteed for any specific problem classes?

Lemma 6.7.4 (Compactness Implies Finite Fixed Set)**.** *If $\mathcal{B}$ is a compact specification, then for any $\mathcal{B}$-unsatisfiable clause set $\mathcal{N}$ there is a defining map M^U with over-approximation set $\mathcal{N}^+$ that is $\mathcal{B}$-unsatisfiable, and M^U contains finitely many definitions.*

Proof. If clause set $\mathcal{N}$ is $\mathcal{B}$-unsatisfiable, then there is a finite subset U of $\mathrm{sgi}(\mathcal{N})$ such that $U \cup \mathrm{GndTh}(\mathcal{B})$ is unsatisfiable. Let M^U consist of $\{t \approx \alpha_t : t \in \mathrm{rel}U$ and α_t is a fresh parameter $\}$ along with definitions for any maximal free BG-sorted terms of $\mathcal{N}$ appropriately constrained to satisfy Definition 6.7.2. As both U and the set of maximal free BG-sorted terms are finite, M^U is finite too. □

Hypothesis: (Under assumption) If given a $\mathcal{B}$-unsatisfiable clause set $\mathcal{N}$ and assuming each call to saturate terminates, then $\mathrm{checkSAT}_{gen}$ eventually terminates with result '$\mathcal{B}$-unsatisfiable'.

This reduces to the condition that every definition $t\sigma$ in $\mathsf{fixed}(\mathrm{M}^U)$ is eventually added by $\mathsf{find}()$, or, conversely, there is no infinite set of term instances outside $\mathsf{fixed}(\mathrm{M}^U)$ selected before an instance in $\mathsf{fixed}(\mathrm{M}^U)$.

Example 6.7.3 (Unfair Update Selection)**.** The unit clause $f(x) < x$ will be unsatisfiable for any interpretation of the form $f(0) \approx \alpha_0, f(1) \approx \alpha_1, \ldots, f(x) \approx \alpha \vee \neg(x \notin [0, n])$, although it is satisfiable on its own. If $f(x) < x$ were part of an overall unsatisfiable clause set, then it could 'hijack' the satisfiability search by generating the infinite sequence of exceptions $f(0), f(1), \ldots$ without reaching M^U.

This is very similar to the fairness condition for saturation based calculi, and a similar condition can be used here.

Recall that an *update heuristic* is a map from labelled (constrained) clauses to a definition and instances of that definition. Specifically, the result of an application of the heuristic is a non-ground definition from the defining map and a substitution which identifies an instance of that definition. The input set of labelled clauses is $\mathcal{B}$-unsatisfiable and is the result of an *HSP* derivation in which labels correspond to the definitions used, and the substitutions used in the derivation of each clause are applied to the label and constraint.

Then a fair update heuristic for M_T should not delay selection of any given term in T, no matter the input. The following definition captures this.

Definition 6.7.4 (Fair Update Heuristic)**.** An update heuristic (i. e. , a find implementation) is *fair* just when for any given ground free BG term $t \in \mathrm{M}_T$ there does not exist a sequence of labelled clause sets $\mathcal{C}_0, \mathcal{C}_1, \ldots$ such that for all $\mathcal{C}_i$

1. in $\mathcal{C}_i$ there is a clause with label $t' \approx \alpha$, constraint D containing μ where $t'\mu = t$; and
2. $\mathsf{find}(\mathcal{C}_i) = (s, \sigma)$ where $s\sigma \neq t$.

This will correct the problem illustrated by Example 6.7.3 only if it is the case that every $t \in \mathsf{fixed}(\mathrm{M}^U)$ eventually appears in a label and constraint pair in a proof of $\mathcal{B}$-unsatisfiability from $\mathcal{N}^-$. This is not guaranteed, and requires an extra assumption on $\mathrm{checkSAT}_{gen}$:

Assumption: Every $t \in \mathsf{fixed}(\mathrm{M}^U)$ is considered by find infinitely often.

This can be accomplished by periodically (say every 5^{th} iteration) sending M_T to find (where definitions label themselves), i. e. , allowing *any* term to be selected.

Theorem 6.7.1. *Let $\mathcal{B}$ be a compact specification and assume that* checkSAT$_{gen}$ *implements a strategy that satisfies the assumption above. If given a $\mathcal{B}$-unsatisfiable clause set $\mathcal{N}$, then, assuming each call to* saturate *terminates and* find *is a fair update heuristic, it follows that* checkSAT$_{gen}$ *eventually terminates with result '$\mathcal{B}$-unsatisfiable'.*

Proof. By compactness and Lemma 6.7.4 M^U exists. By assumption, the entire set of definition terms in $\mathrm{M}_{\mathrm{rel}(\mathcal{N})}$ is eligible for selection infinitely often. Since find is fair, any $t \in \mathsf{fixed}(\mathrm{M}^U)$ is eventually selected for updating, and so the particular unsatisfiable over-approximation $\mathcal{N}^+$ is eventually tested. By Lemma 6.7.3, none of the $\mathcal{N}^-$ sets tested prior to that incorrectly conclude '$\mathcal{B}$-satisfiable', and by Lemma 6.7.1, testing $\mathcal{N}^+$ returns the required '$\mathcal{B}$-unsatisfiable' result. □

It remains to show that a fair heuristic exists. The heuristics mentioned in Section 6.3.2 minimize w. r. t. cardinality of the update only, so are not necessarily fair.

Example 6.7.4 (A Fair Update Heuristic)**.** Consider the heuristic that on input $\mathcal{C}_i$ returns a ground term instance $t_0\gamma_0, t_1\gamma_1, \ldots$ according to the rules:

1. Select $t_0\gamma_0$ from $\mathcal{C}_0$ arbitrarily, store t_0 as p.
2. If there is a term t_{i+1} in the labels of $\mathcal{C}_{i+1}$ such that $t_{i+1} \prec t_i$ according to the term order, select that set $p = t_{i+1}$.
3. Otherwise, select t_{i+1} from $\mathcal{C}_{i+1}$ arbitrarily. If there are no terms smaller than t_{i+1} in $\mathsf{ge}(\mathrm{M}_T)$ and not in $\mathsf{fixed}(\mathrm{M}_T)$, then set $p = t_{i+1}$.

The assumption on checkSAT$_{gen}$ strategy ensures that option (3) is never chosen infinitely often in sequence without updating p. The phrase 'select from $\mathcal{C}_i$' means choose a definition $(t \approx \alpha)$ labelling a constrained clause $C \mid D$ and a substitution $\gamma \in D$. Since the term order is well-founded, it follows that only finitely many terms are selected in option (2) before option (3) is chosen and p is updated. Therefore the given heuristic is fair.

The heuristic in Example 6.7.4 only returns ground instances as updates. It may be impossible to have heuristics that return non-ground instances for the general case, since they are generally not ordered.

As observed in Section 5.6.1, for fragments in which satisfiability can be decided by testing a finite set of clause instances, it may be enough to give definitions for just the relevant terms found in that finite instantiation. In that case, the set $\mathsf{fixed}(M^U)$ is known, and a fair heuristic could simply ensure that those terms are eventually added, meanwhile returning whatever terms it chooses. Thus, more efficient updates can be returned, while guaranteeing completeness overall. Such considerations remain to be verified, however.

Ideas from this could be used inside the *HSP* calculus: already the Define rule is applied eagerly to recover sufficient completeness. In addition, instantiation could be used to introduce new definitions for instances of relevant terms produced via E-matching rather than unification. This sacrifices the possible efficiency gains from

using non-ground definitions, and doesn't guarantee sufficient completeness, but it avoids the usual problem in using instantiation, namely, that new clause instances are immediately eligible for simplification by subsumption.

6.8 Summary

The given hierarchic satisfiability algorithm $\mathrm{checkSAT_M}$ augments a prover for first-order logic modulo theories by recovering completeness when the set of relevant terms is finite. Strong theory reasoning capabilities, specifically for linear integer arithmetic, enable an intensional description of sets of relevant terms, which is exploited in the hope of avoiding excessive instantiation– anathema to first-order solvers. However, this is the usual solution when reasoning over fragments in which satisfiability is equal to satisfiability in a finite set of instances [GdM09, ISS09].

The only way to exploit that compact description is to use default (parametric) values, necessitating an under- then over-approximation approach to reasoning, similar to that described by Lynch [Lyn04]. In this approach, a simplifying constraint on the equational structure of possible interpretations is hypothesized and then iteratively refined. Some heuristics are required to avoid performing more solver calls than would be done if outright instantiation were used instead.

By focusing on the free BG sorted terms rather than the finite domains as the means for organizing definitions, performance advantages were obtained over the original hierarchic satisfiability procedure for modular solvers [BW13b].

Clause labels were used to produce a smaller set of possible repairs at the end of an unsuccessful test of a particular defining map. Though requiring modification of the component solver, this technique avoids repeating many similar proofs, as happens in the original find method that was based on binary search.

Although smaller, the final set of labels is usually not definitive, and so several heuristics were given, and compared against the original implementation. Two of the heuristics (Z3-MUC and $\mathrm{NG{+}red}$) require the finite quantification restriction, but one method ($\mathrm{NG\text{-}MUC}$) does not, and that could be adapted for other use cases, such as where quantifier ranges are unbounded or over non-integer sorts.

The general form of the algorithm was also specialized to recursive data structure domains. Theoretical results where given, though most problems over these domains require a component solver with better inductive reasoning capability to handle the $\mathrm{saturate}()$ calls.

The description of basic definitions allows automatic recognition of relevant terms that can be excluded from the defining map, thereby improving efficiency. This fixes a problem left implicit in [BW13b].

One further way of generalizing the method is to move to an incomplete search, lifting the requirement for finite quantification of variables in relevant terms and defining maps. This would be similar to the heuristic quantifier instantiation methods found in SMT [dMB07], or to instantiation based first-order reasoning (for example, Ganzinger and Korovin [GK04b]) when in the over-approximation phase.

In particular, dismatching constraints are used to prevent trivial definitions, and E-matching is used to discover useful updates when the NG-MUC heuristic is not effective.

Implementation and testing of this variation on instance-based reasoning is future work.

Chapter 7

Conclusion

This thesis describes some techniques for first-order reasoning with theories, focussing in particular on those which enable a first-order reasoner to conclude satisfiability of a formula, modulo an arithmetic theory. These new techniques form a useful complement to existing methods that are primarily aimed at proving validity, though unsatisfiable problems remain the easiest to solve.

Each of these make use of the theory reasoning capability of the Hierarchic Superposition calculus, combining equational reasoning with native support for quantifiers and building in decision procedures for arithmetic theories. Weak abstraction and related improvements make an implementation of the calculus feasible.

The first contribution is an implementation of that calculus (*Beagle*), including an optimized implementation of Cooper's algorithm for quantifier elimination in the theory of linear integer arithmetic. This includes a novel means of extracting certain values for quantified variables in satisfiable integer problems with arbitrary quantification. In addition, *Beagle* includes theory solvers for rational and real linear arithmetic as well as an interface (via SMT-lib) to compatible SMT solvers. *Beagle* accepts input in both SMT-lib and TPTP format, meaning it can interface with verification tools like the Why3 intermediate verification language and the Sledgehammer solver for Isabelle/HOL. *Beagle* won an efficiency award at CASC-J7, and won the arithmetic non-theorem category at CASC-25. This implementation is the start point for solving the 'disproving with theories' problem.

The first satisfiability method, and the first use of definitions, gives syntactic criteria for recognising when an unsatisfiable formula implies satisfiability of a particular subformula: the hypothesis. If the input formula is divided into satisfiable (known) axioms and satisfiability preserving definitions that extend the axioms, the remainder must be the cause of the unsatisfiability, and therefore has no model relative to the axioms. These syntactic criteria include well-founded recursive definitions, definitions over lists, and to arrays. This allows proving some non-theorems which are otherwise intractable, and justifies similar disproofs of non-linear arithmetic formulas. Using these results, a selection of non-theorems shown satisfiable, where the corresponding negated forms (counter-satisfiable) were unable to be solved.

When the hypothesis is contingently true, disproof requires proving existence of a model. If the Superposition calculus saturates a clause set, then a theory extending model exists, but only when the clause set satisfies a completeness criterion. This

requires each instance of an uninterpreted theory-sorted term to have a definition in terms of theory symbols. If that were not the case, then any model found may not properly extend the background theory, meaning it is not correct to conclude satisfiability.

The method described in Chapters 5 and 6, checkSAT requires that certain quantifiers are restricted to range over finite sets, and builds definitions for those uninterpreted theory-sorted terms. Moreover, the use of first-order reasoning allows for an implicit representation of those finite sets, possibly avoiding scalability problems that affect other quantifier reasoning methods.

Definitions are produced in a counter-example driven way via a sequence of over and under approximations to the clause set. Two descriptions of the method are given: the first uses the component solver modularly, but has an inefficient counter-example heuristic. The second is more general, correcting many of the inefficiencies of the first, yet it requires tracking clauses through a proof. This latter method is shown to apply also to lists and to problems with unbounded quantifiers. Furthermore, the recognition of *basic definitions* already present in the input formula allows for reducing the number of terms that need to be defined using the checkSAT method, improving overall efficiency.

Lastly, a sketch proof is given for how the checkSAT method could be used for clause sets without finite domains. Although only refutation completeness is possible there, this nevertheless extends the capabilities of the basic Hierarchic Superposition calculus, as it is not guaranteed to be refutation complete in the absence of sufficient completeness.

Together, these tools give new ways for applying successful first-order reasoning methods to problems involving interpreted theories.

7.1 Future Work

As with all software described herein, *Beagle* is prototypical and lacks many state-of-the-art features found in other solvers. Specifically, performance is rather poor on large formulas, or on formulas with large boolean components (e. g. shallow terms, many boolean variables). Improvements to term-indexing would likely improve the situation, as would more powerful simplification strategies. Proofs can be generated for the equational part of a derivation, however there is no proof procedure for Cooper's algorithm (the aforementioned quantifier value extraction method only returns values for the innermost quantifiers).

The syntactic test for admissible definitions could be automated and integrated into *Beagle* , although initial tests suggest it is difficult to cover all variations. Perhaps it would be better used as a general solver strategy that relies on the user specifying definitions, e. g. using the SMT-lib input language. The observation that these types of formulas are satisfiability preserving could be used in simplification strategies, similar to how Armando et al. [ABRS09] restrict inferences between theory axioms.

In addition, an automated method of finding bounded domains, or mapping

from finite domains appropriately would be necessary to apply $\mathrm{checkSAT}$ at a large scale. Refinements to the way definitions are stored and manipulated would also improve performance; the formal methods literature is replete with methods for representing substitution sets and integer partitions efficiently. Further experimentation with various component solvers in $\mathrm{checkSAT}$, or using an SMT solver as an oracle for constructing models would also be very interesting.

Finally, the refutation complete, unbounded algorithm $\mathrm{checkSAT}_{gen}$ warrants expansion, especially as there are few methods exploring this style of theorem proving in the first-order reasoning literature. It has been remarked to me that Constraint Satisfaction only came into its own once the field started exploring incomplete heuristics, perhaps the same is true of automated reasoning?

References

ABRS09. Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1), 2009. (cited on pages 8, 16, 17, 72, 77, 90, 136, 137, and 148)

AKW09. Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. Superposition modulo linear arithmetic SUP(LA). In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, volume 5749 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2009. (cited on pages 32, 90, and 136)

Bau15. Peter Baumgartner. SMTtoTPTP – a converter for theorem proving formats. volume 9195 of *Lecture Notes in Computer Science*, pages 285–294. Springer, 2015. (cited on page 63)

BB13. Peter Baumgartner and Joshua Bax. Proving infinite satisfiability. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 68–95. Springer, 2013. (cited on pages 3 and 81)

BBW14. Peter Baumgartner, Joshua Bax, and Uwe Waldmann. Finite quantification in hierarchic theorem proving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2014. (cited on pages 3, 37, 89, 100, 114, and 132)

BBW15. Peter Baumgartner, Joshua Bax, and Uwe Waldmann. Beagle – A Hierarchic Superposition Theorem Prover. volume 9195 of *Lecture Notes in Computer Science*, pages 367–377. Springer, 2015. (cited on pages 3 and 37)

BC96. Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In Hélène Kirchner, editor, *Trees in Algebra and Programming — CAAP '96: 21st International Colloquium Linköping. Proceedings*, pages 30–43. Springer Berlin Heidelberg, 1996. (cited on page 15)

BCD+05. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. (cited on page 8)

BCD+11. Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. (cited on page 68)

BFdNT09. Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 7(1):58–74, 2009. (cited on pages 91 and 110)

BFT15. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`. (cited on page 85)

BG94. Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. (cited on pages 20 and 25)

BG98. Leo Bachmair and Harald Ganzinger. Equational reasoning in saturation-based theorem proving. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction: A Basis for Applications*, pages 353–397. Kluwer, 1998. (cited on pages 5 and 6)

BGW94. Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3/4):193–212, April 1994. (cited on pages 7, 22, 23, 26, 27, 28, 30, 37, and 90)

BH96. Arnim Buch and Thomas Hillenbrand. Waldmeister: Development of a high performance completion-based theorem prover. Technical Report SR-96-01, Universit at Kaiserslautern, 1996. (cited on page 6)

BHZ05. Lucas Bordeaux, Youssef Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. Technical Report MSR-TR-2005-124, Microsoft, 2005. (cited on page 4)

BKKS13. Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. On verification by translation to recursive functions. Technical Report 186233, EPFL, 2013. (cited on page 111)

BM07. Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer, 2007. (cited on pages 8, 13, 14, 15, and 16)

BMS06. Aaron R Bradley, Zohar Manna, and Henny B Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006. (cited on pages 14, 16, 87, and 110)

BN98. F. Baader and T. Nipkow. *Term Rewriting and all that*. Cambridge University Press, Cambridge, 1998. (cited on pages 20 and 21)

BN10. Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving: First International Conference, ITP 2010*, pages 131–146. Springer, 2010. (cited on page 35)

BPSV12. Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012. (cited on page 8)

Bra75. D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4:412–430, 1975. (cited on page 20)

BS09. Peter Baumgartner and John Slaney. Constraint modelling: A challenge for automated reasoning. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 7th International Workshop on First-Order Theorem Proving (FTP'09)*, volume 556 of *Workshop Proceedings*, pages 4–18. CEUR, 2009. (cited on page 5)

BSST09. Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. (cited on pages 8 and 32)

BST10. Clark Barrett, A. Stump, and Cesare Tinelli. The SMT-LIB standard - version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland,(SMT '10)*, 2010. (cited on pages 37 and 62)

BT05. Peter Baumgartner and Cesare Tinelli. The model evolution calculus with equality. volume 3632 of *Lecture Notes in Computer Science*, pages 392–408. Springer, 2005. (cited on page 120)

BT11. Peter Baumgartner and Cesare Tinelli. Model evolution with equality modulo built-in theories. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International*

Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings, volume 6803 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2011. (cited on page 90)

Bür94. Hans-Jürgen Bürckert. A resolution principle for constrained logics. *Artificial intelligence*, 66(2):235–271, 1994. (cited on page 7)

BW13a. Peter Baumgartner and Uwe Waldmann. Hierarchic superposition: Completeness without compactness. In Marek Kosta and Thomas Sturm, editors, *Mathematical Aspects of Computer and Information Sciences - 5th International Conference, MACIS 2013*, pages 8–12, 2013. (cited on pages 29, 90, and 136)

BW13b. Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. volume 7898 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2013. (cited on pages 9, 22, 23, 24, 26, 27, 28, 29, 37, 63, 72, 90, 137, and 145)

CJ98. Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998. (cited on page 15)

CL11. Koen Claessen and Ann Lilliesträm. Automated inference of finite unsatisfiability. *Journal of Automated Reasoning*, 47(2):111–132, 2011. (cited on page 87)

Coo72. D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier. (cited on pages 14, 15, and 49)

CS03. Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model building. In Peter Baumgartner and Christian G. Fermüller, editors, *CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications*, 2003. (cited on pages 63, 91, and 110)

Dec03. Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003. (cited on page 4)

Den00. Marc Denecker. Extending classical logic with inductive definitions. In *Computational Logic-CL 2000*, pages 703–717. Springer, 2000. (cited on page 75)

Der82. N. Dershowitz. Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, 17:279–301, 1982. (cited on page 21)

dMB07. Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007. (cited on pages 90, 110, and 145)

dMB08. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. (cited on pages 42, 43, 65, 109, and 132)

dMB09. Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 45–52. IEEE, 2009. (cited on page 17)

DNS03. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003. (cited on pages 8 and 33)

Dow72. P Downey. Undecidability of presburger arithmetic with a single monadic predicate letter. Technical report, Havard University, 1972. (cited on pages 15 and 90)

FP13. Jean-Christophe Filliâtre and Andrei Paskevich. Why3âĂŤwhere programs meet provers. In *Programming Languages and Systems*, pages 125–128. Springer, 2013. (cited on page 8)

FR74. Michael J Fischer and Michael O Rabin. Super-exponential complexity of Presburger arithmetic. In *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 27–42. AMS, 1974. (cited on page 15)

GBT07. Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2007. (cited on pages 8, 90, and 110)

GdM09. Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009. (cited on pages 8, 17, 33, 87, 90, 110, 111, 113, and 145)

GK03. H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proc. 18th IEEE Symposium on Logic in Computer Science,(LICS'03)*, pages 55–64. IEEE Computer Society Press, 2003. (cited on page 111)

GK04a. H. Ganzinger and K. Korovin. Integrating equational reasoning into instantiation-based theorem proving. In *Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2004. (cited on page 111)

GK04b. H. Ganzinger and K. Korovin. Integrating equational reasoning into instantiation-based theorem proving. In *Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2004. (cited on pages 140 and 145)

GK06. H. Ganzinger and K. Korovin. Theory Instantiation. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2006. (cited on page 90)

GNRZ07. Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Decision procedures for extensions of the theory of arrays. *Annals of Mathematics and Artificial Intelligence*, 50(3-4):231–254, 2007. (cited on pages 17 and 87)

Haa14. Christoph Haase. Subclasses of Presburger arithmetic and the weak EXP hierarchy. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 47. ACM, 2014. (cited on page 15)

Hal91. Joseph Y. Halpern. Presburger arithmetic with unary predicates is Π^1_1 complete. *Journal of Symbolic Logic*, 56:637–642, 1991. (cited on pages 15 and 90)

Har09. John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. (cited on pages 15, 47, 49, 50, and 56)

HSS13. Matthias Horbach and Viorica Sofronie-Stokkermans. Obtaining finite local theory axiomatizations via saturation. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2013. (cited on page 19)

HW07. Thomas Hillenbrand and Christoph Weidenbach. Superposition for finite domains. Research Report MPI-I-2007-RG1-002, Max-Planck Institute for Informatics, Saarbruecken, Germany, April 2007. (cited on pages 95 and 97)

HW13. Thomas Hillenbrand and Christoph Weidenbach. Superposition for bounded domains. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 68–100. Springer, 2013. (cited on page 97)

IJSS08. Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2008. (cited on pages 17 and 110)

ISS09. Carsten Ihlemann and Viorica Sofronie-Stokkermans. System description: H-PILoT. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 131–139. Springer, 2009. (cited on page 145)

ISS10. Carsten Ihlemann and Viorica Sofronie-Stokkermans. On hierarchical reasoning in combinations of theories. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2010. (cited on page 19)

JB11. Dejan Jovanović and Clark Barrett. Sharing is caring: Combination of theories. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2011. (cited on page 33)

KB83. Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983. (cited on pages 6 and 20)

KKP+15. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015. (cited on page 8)

KNZ87. Deepak Kapur, Paliath Narendran, and Hantao Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, 1987. (cited on page 29)

Kor13. Konstantin Korovin. Non-cyclic sorts for first-order satisfiability. volume 7898 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2013. (cited on page 23)

KOSS04. Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 308–320. Springer, 2004. (cited on page 15)

KV07. 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. (cited on pages 32 and 90)

KV13. L. Kovacs and A. Voronkov. First-Order Theorem Proving and Vampire. volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013. (cited on page 69)

KW12. Evgeny Kruglov and Christoph Weidenbach. Superposition decides the first-order logic fragment over ground theories. *Mathematics in Computer Science*, pages 1–30, 2012. (cited on pages 29 and 90)

KZ05. Deepak Kapur and Calogero G Zarba. A reduction approach to decision procedures. Technical Report TR-CS-2005-44, University of New Mexico, 2005. (cited on pages 17 and 18)

Lei10. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. (cited on page 8)

Lyn04. Christopher Lynch. Unsound theorem proving. In Jerzy Marcinkowski and Andrej Tarlecki, editors, *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2004. (cited on pages 111 and 145)

Mac92. Alan K Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58(1):3–20, 1992. (cited on pages 4 and 5)

McC62. J. McCarthy. Towards a mathematical theory of computation. In *Proceedings of IFIP Congress*, pages 21âĂŞ–28, 1962. (cited on pages 13 and 16)

McC03. W. McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, 2003. (cited on pages 91 and 110)

Mon10. David Monniaux. Quantifier elimination by lazy model enumeration. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2010. (cited on pages 15, 45, and 61)

Nie10. Robert Nieuwenhuis. Sat modulo theories: Getting the best of sat and global constraint filtering. In David Cohen, editor, *Principles and Practice of Constraint Programming âĂŞ CP 2010*, volume 6308 of *Lecture Notes in*

Computer Science, pages 1–2. Springer Berlin Heidelberg, 2010. (cited on pages 5 and 8)

NO79. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979. (cited on pages 8 and 33)

NO80. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of Association for Computer Machinery*, 27(2), April 1980. (cited on page 14)

NOT06. 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 Association for Computing Machinery*, 53(6):937–977, 2006. (cited on pages 8 and 90)

NR01. Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001. (cited on pages 5 and 20)

Opp78. Derek C Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, 1978. (cited on page 15)

Opp80. Derek C Oppen. Reasoning about recursively defined data structures. *Journal of the Association for Computing Machinery*, 27(3):403–411, 1980. (cited on pages 18, 19, and 137)

PH15. Anh-Dung Phan and Michael R Hansen. An approach to multicore parallelism using functional programming: A case study based on presburger arithmetic. *Journal of Logical and Algebraic Methods in Programming*, 84(1):2–18, 2015. (cited on pages 15 and 51)

PJ91. MojÅijesz Presburger and Dale Jabcquette. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12(2):225–233, 1991. (cited on page 14)

Pug91. William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 4–13. ACM, 1991. (cited on pages 15 and 49)

RBCT16. Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model finding for recursive functions in SMT. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International*

Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings, volume 9706 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2016. (cited on page 87)

RL78. Cattamanchi R Reddy and Donald W Loveland. Presburger arithmetic with bounded quantifier alternation. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 320–325. ACM, 1978. (cited on page 15)

Rob65a. J. A. Robinson. Automated deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1(3):227–234, 1965. (cited on page 65)

Rob65b. J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965. (cited on page 5)

RS11. Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2011. (cited on page 131)

RTG+13. Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013. (cited on pages 34, 91, 110, and 120)

RTGK13. Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite model finding in SMT. volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer, 2013. (cited on pages 34, 91, 95, 109, 110, and 111)

Rüm08. Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2008. (cited on pages 34, 68, and 90)

Rüm12. Philipp Rümmer. E-matching with free variables. volume 7180 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2012. (cited on page 34)

RV01. Alexandre Riazonov and Andrei Voronkov. Vampire 1.1 (system description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2001. (cited on page 6)

RW69. G. A. Robinson and L. Wos. Paramodulation and Theorem Proving in First Order Theories with Equality. In Meltzer and Mitchie, editors, *Machine Intelligence 4*. Edinburg University Press, 1969. (cited on page 5)

Sch04. S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004. (cited on page 6)

Sch13. Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, pages 45–67. Springer Berlin Heidelberg, 2013. (cited on page 65)

SDK10. Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. *ACM SIGPLAN Notices*, 45(1):199–210, 2010. (cited on page 18)

SKK11a. Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 298–315. Springer, 2011. (cited on pages 18 and 111)

SKK11b. Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In Eran Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 298–315. Springer, 2011. (cited on page 87)

SKR98. Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. A comparison of Presburger engines for EFSM reachability. volume 1427 of *Lecture Notes in Computer Science*, pages 280–292. Springer, 1998. (cited on page 15)

Sla92. John Slaney. Finder (finite domain enumerator): Notes and guide. Technical Report TR-ARP-1/92, Australian National University, Automated Reasoning Project, Canberra, 1992. (cited on pages 91 and 110)

SS05. Viorica Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. volume 3632 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2005. (cited on pages 18, 19, and 30)

SSCB12. Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP typed first-order form with arithmetic. volume 7180 of *Lecture Notes in Computer Science*, pages 406–419. Springer, 2012. (cited on page 62)

Sti85. M.E. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1:333–355, 1985. (cited on page 7)

SU16. Geoff Sutcliffe and Josef Urban. The CADE-25 automated theorem proving system competition–CASC-25. *AI Communications*, 29(3):423–433, 2016. (cited on page 68)

Sut09. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009. (cited on pages 37 and 64)

Sut14. Geoff Sutcliffe. The CADE-24 automated theorem proving system competition–CASC-24. *AI Communications*, 27(4):405–416, 2014. (cited on page 34)

Sut15. Geoff Sutcliffe. The 7th IJCAR automated theorem proving system competition–CASC-J7. *AI Communications*, 28(4):1–10, 2015. (cited on pages 34 and 68)

Sut16. Geoff Sutcliffe. The 8th IJCAR automated theorem proving system competition–CASC-J8. *AI Communications*, 29(5):607–619, 2016. (cited on pages 66 and 69)

TCJ08. Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008. (cited on page 131)

TJ07. Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007. (cited on page 35)

VB14. Andrei Voronkov and Roderick Bloem. AVATAR: The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014. (cited on page 95)

Woo15. Kevin Woods. Presburger arithmetic, rational generating functions, and quasi-polynomials. *Journal of Symbolic Logic*, 80(02):433–449, 2015. (cited on page 15)

WP06. Uwe Waldmann and Virgile Prevosto. SPASS+T. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Empirically Successful Computerized Reasoning (ESCoR)*, volume 192 of *CEUR Workshop Proceedings*, pages 18–33, 2006. (cited on pages 27, 34, and 68)

WSH+07. Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Rostislav Rusev, and Dalibor Topic. System description: Spass version 3.0. In Frank Pfenning, editor, *CADE-21 — 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 514–520. Springer, 2007. (cited on page 6)

ZSM04. Ting Zhang, Henny B Sipma, and Zohar Manna. Decision procedures for recursive data structures with integer constraints. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2004. (cited on page 18)

ZZ95. Jian Zhang and Hantao Zhang. Sem: a system for enumerating models. In Chris Mellish, editor, *IJCAI-95 — Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal*, pages 298–303. Morgan Kaufmann, 1995. (cited on pages 91 and 110)