# Decision Procedures for Finite Sets with Cardinality and Local Theory Extensions

by

Kshitij Bansal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
January 2016

---

Clark W. Barrett

---

Thomas Wies

# Acknowledgements

I'd like to thank my advisors Clark Barrett and Thomas Wies for their guidance and support. I'd also like to thank Subhash Khot and Stéphane Demri for their guidance during my PhD. I'd like to thank the members of the CVC4 team, especially Morgan Deters, without whom this work would not have been possible. I'd like to thank Ruzica Piskac, Viktor Kuncak, Cesare Tinelli, and Stéphane Demri for inviting me to their universities and many engaging discussions during the visits. I'd like to thank my collaborators Andrew Reynolds, Tim King, Liana Hadarean, Dejan Javonović, Eric Koskinen, Cesare Tinelli, Omer Tripp, and Damien Zufferey. I'd like to thank the members of my thesis committee for their time and valuable feedback: Clark Barrett, Thomas Wies, Benjamin Goldberg, Eric Koskinen, and Cesare Tinelli. I'd also like to thank Madhavi Saxena, Madhavan Mukund, Narayan Kumar and all the excellent teachers I had the fortune of learning from. Finally, I'd like to thank my parents and my brother, Ankur, for their unwavering love and support.

# Abstract

Many tasks in design, verification, and testing of hardware and computer systems can be reduced to checking satisfiability of logical formulas. Certain fragments of first-order logic that model the semantics of prevalent data types, and hardware and software constructs, such as integers, bit-vectors, and arrays are thus of most interest. The appeal of satisfiability modulo theories (SMT) solvers is that they implement decision procedures for efficiently reasoning about formulas in these fragments. Thus, they can often be used off-the-shelf as automated back-end solvers in verification tools. In this thesis, we expand the scope of SMT solvers by developing decision procedures for new theories of interest in reasoning about hardware and software.

First, we consider the theory of finite sets with cardinality. Sets are a common high-level data structure used in programming; thus, such a theory is useful for modeling program constructs directly. More importantly, sets are a basic construct of mathematics and thus natural to use when mathematically defining the properties of a computer system. We extend a calculus for finite sets to reason about cardinality constraints. The reasoning for cardinality involves tracking how different sets overlap. For an efficient procedure in an SMT solver, we'd like to avoid considering Venn regions directly, which has been the approach in earlier work. We develop a novel technique wherein potentially overlapping regions are considered incrementally. We

use a graph to track the interaction of the different regions. Additionally, our technique leverages the procedure for reasoning about the other set operations (besides cardinality) in a modular fashion.

Second, a limitation frequently encountered is that verification problems are often not fully expressible in the theories supported natively by the solvers. Many solvers allow the specification of application-specific theories as quantified axioms, but their handling is incomplete outside of narrow special cases. We show how SMT solvers can be used to obtain complete decision procedures for local theory extensions, an important class of theories that are decidable using finite instantiation of axioms. We present an algorithm that uses E-matching to generate instances incrementally during the search, significantly reducing the number of generated instances compared to eager instantiation strategies.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Consider some curiosities of an idling mind:

- Sam is 5 years older than Ida. Mir is thrice the age of Ida. Sam is twice the age of Mir. Is that possible?

- Is it possible to place 8 queens on a (standard, 8-by-8) chessboard such that no two of them attack each other?

- Mia came across the following function in the C programming language:

```
int f(int x) {
  if( x < 0 ) {
    x = -x;
  }
  return 1000 / (1 + x);
}
```

  She wonders, can this function ever have a divide-by-zero error?

- A plane is flying east at an altitude of 20,000 ft and speed of 400 miles per hour. Another plane, east of the first plane is flying west towards it at the same altitude and the same speed. When 2 km apart, one of the pilots realizes they are headed for a head-on collision, and establishes contact with the other plane. Given constraints on the maneuvering capabilities of the planes, can the pilots navigate past each other maintaining a safe distance?

The above quite disparate looking problems can be viewed to be of the same form: they all impose some constraints on the possibilities, and ask if a solution exists. For instance, in the case of the first problem, a natural approach to solve involves the following two distinct steps. One, encode the problem in a mathematically precise way. E.g. let age of Sam be $x$, and that of Ida be $y$, then it must be the case that $x = y + 5$. Two, once we have a set of constraints, does that have a solution?

Naturally, we are interested in using computers to aid us in solving these, but the process is be the same: one, encoding the problem as constraints in a "language with precisely defined meaning" and two, solving the constraints. Indeed, many problems in computer science (and, in other domains) can be reduced to checking the satisfiability of constraints encoded using logical formulas ("language with precisely defined meaning"). The focus of this thesis is on the second step of determining if a set of constraints has a solution.

There has been tremendous growth of techniques to check satisfiability of logical formulas, and these techniques have applications in several domains of hardware and software engineering. Applications include equivalence checking, model-checking, assertions checking, and test pattern generation in hardware engineering; and dynamic symbolic execution, program model checking, static model checking, program verification, program synthesis and software modeling in software engineering.

In order to move any further, we need to fix a language and an unambiguous meaning for it. We begin by doing so, which will then allow us to describe the content and contribution of the thesis.

## 1.1 First-order logic

### 1.1.1 Syntax

Given a finite set of *sorts* Sorts, a *signature* for first-order logic is a tuple $(\mathsf{Funcs}, \mathsf{Preds})$ where Funcs is a set of *function symbols* and Preds is set of *predicate symbols*. Each function and predicate symbol has an arity and a rank associated with it. The *arity* is a non-negative integer. The *rank of a function symbol* of arity $n$ is described as $s_1 \times s_2 \ldots \times s_n \rightarrow s_{n+1}$ where each $s_i \in \mathsf{Sorts}$. The *rank of a predicate symbol* of arity $n$ is described as $s_1 \times \ldots \times s_n$ where each $s_i \in \mathsf{Sorts}$. We sometimes refer to a function symbol with arity $n$ as *$n$-ary*. A *constant symbol* is a $0$-ary function symbol. A *propositional symbol* is a $0$-ary predicate symbol. We assume a countably infinite set of *variable symbols* for each sort.

Let $\Sigma = (\mathsf{Funcs}, \mathsf{Preds})$ be a signature over sorts Sorts. Then, a *$\Sigma$-term* of sort $s$ is defined as one of the following:

1. a variable symbol of sort $s$,
2. a constant symbol in Funcs of sort $s$, or
3. $f(t_1, \ldots, t_n)$ where $f$ in Funcs is an $n$-ary function symbol of rank $s_1 \times s_2 \ldots \times s_n \rightarrow s$, and $t_1, t_2 \ldots t_n$ are $\Sigma$-terms of sorts $s_1, s_2 \ldots s_n$ respectively.

A *$\Sigma$-formula* is defined as one of the following:

1. $t_1 = t_2$ where $t_1$ and $t_2$ are terms of the same sort,

2. $P(t_1, \ldots, t_n)$ where $P$ in Preds is an $n$-ary predicate symbol of rank $s_1 \times s_2 \ldots \times s_n$, and $t_1$, $t_2 \ldots t_n$ are $\Sigma$-terms of sorts $s_1$, $s_2 \ldots s_n$ respectively,

3. $\neg\phi$ where $\phi$ is a $\Sigma$-formula,

4. $\phi_1 * \phi_2$ where $*$ is a symbol from the set $\{\vee, \wedge, \rightarrow, \leftrightarrow\}$ and $\phi_1$ and $\phi_2$ are $\Sigma$-formulas,

5. $\forall v\ \phi$ where $v$ is a variable symbol and $\phi$ is a $\Sigma$-formula,

6. true, or

7. false.

A $\Sigma$-*atomic proposition* is a formula of the form 1 or 2.

### 1.1.2 Semantics

In order to give meaning to formulas, we introduce the notion of structures, and use it to interpret terms and formulas.

As earlier, let $\Sigma = (\mathsf{Funcs}, \mathsf{Preds})$ be a signature over sorts Sorts. Then, a $\Sigma$-*structure* $\mathfrak{I}$ consists of:

- for each $s \in$ Sorts: a set $\mathcal{U}_s$ called the domain of sort $s$,
- for each $f \in$ Funcs of arity $n$ and rank $s_1 \times s_2 \ldots \times s_n \rightarrow s_{n+1}$: a function $f^{\mathfrak{I}} : \mathcal{U}_{s_1} \times \ldots \times \mathcal{U}_{s_n} \rightarrow \mathcal{U}_{s_{n+1}}$, and
- for each $P \in$ Preds of arity $n$ and rank $s_1 \times s_2 \ldots \times s_n$, a function $f^{\mathfrak{I}} : \mathcal{U}_{s_1} \times \ldots \times \mathcal{U}_{s_n} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ where $n$ is the arity of $P$.

A *variable assignment* is a mapping from variable symbols to $\mathcal{U}$.

Given a structure $\mathfrak{I}$, variable assignment $\nu$, and term $t$, we define $t^{\mathfrak{I},\nu}$ as follows:

- $\nu(t)$ if $t$ is a variable,
- $t^{\mathfrak{I}}$ if $t$ is a constant, and
- $f^{\mathfrak{I}}(t_1^{\mathfrak{I},\nu}, \ldots t_n^{\mathfrak{I},\nu})$ if $t$ is $f(t_1, \ldots, t_n)$.

Given a structure $\mathfrak{I}$, variable assignment $\nu$ and formula $\phi$, we define $\mathfrak{I}, \nu \models \phi$ recursively:

- If $\phi$ is $t_1 = t_2$, then $\mathfrak{I}, \nu \models \phi$ iff $t_1^{\mathfrak{I},\nu} = t_2^{\mathfrak{I},\nu}$.
- If $\phi$ is $P(t_1, \ldots, t_n)$, then $\mathfrak{I}, \nu \models \phi$ iff $P^{\mathfrak{I}}(t_1^{\mathfrak{I},\nu}, \ldots t_n^{\mathfrak{I},\nu}) = \mathsf{true}$.
- If $\phi$ is $\neg\phi'$, then $\mathfrak{I}, \nu \models \phi$ iff $\mathfrak{I}, \nu \not\models \phi'$.
- If $\phi$ is $\phi_1 \vee \phi_2$, then $\mathfrak{I}, \nu \models \phi$ iff $\mathfrak{I}, \nu \models \phi_1$ or $\mathfrak{I}, \nu \models \phi_2$.
- If $\phi$ is $\phi_1 \wedge \phi_2$, then $\mathfrak{I}, \nu \models \phi$ iff $\mathfrak{I}, \nu \models \phi_1$ and $\mathfrak{I}, \nu \models \phi_2$.
- If $\phi$ is $\phi_1 \rightarrow \phi_2$, then $\mathfrak{I}, \nu \models \phi$ iff $\mathfrak{I}, \nu \not\models \phi_1$ or $\mathfrak{I}, \nu \models \phi_2$.
- If $\phi$ is $\phi_1 \leftrightarrow \phi_2$, then $\mathfrak{I}, \nu \models \phi$ iff $\mathfrak{I}, \nu \models \phi_1 \rightarrow \phi_2$ and $\mathfrak{I}, \nu \models \phi_2 \rightarrow \phi_1$.
- If $\phi$ is $\forall v\ \phi'$, then $\mathfrak{I}, \nu \models \phi$ iff for all $c \in \mathcal{U}$, $\mathfrak{I} \models_{\nu[v \mapsto c]} \phi'$.
- If $\phi$ is $\mathsf{true}$, then $\mathfrak{I}, \nu \models \phi$.
- If $\phi$ is $\mathsf{false}$, then $\mathfrak{I}, \nu \not\models \phi$.

Given a $\Sigma$-formula $\phi$, we say it is *satisfiable* if there exists a structure $\mathfrak{I}$ and variable assignment $\nu$ such that $\mathfrak{I}, \nu \models \phi$.

Henceforth, we drop the prefix $\Sigma$ where it is clear from the context.

## 1.2 Satisfiability problem

We can now define the satisfiability problem for first-order logic, which will be the focus of the thesis.

**Problem 1.1** (First-order satisfiability)**.** *The satisfiability problem for first-order logic is the decision problem to check if a first-order formula is satisfiable.*

**input:** *A finite set of sorts* Sorts*, a signature* $\Sigma$ *over* Sorts*, and a* $\Sigma$*-formula* $\phi$*.*

**output:** *Yes, if* $\phi$ *is satisfiable. No, otherwise.*

Is there a procedure for the above the decision problem? In 1928, a slightly different formulation of this problem was posed by David Hilbert [28], also famously known as the *Entscheidungsproblem*. In 1936, it was answered in the negative by Alonzo Church* and Alan Turing† under an assumption now commonly known as the Church-Turing thesis.

A natural question then arises, are there meaningful restrictions of the problem which allow us to regain decidability? As an exercise, let us consider a very restrictive fragment of the problem. Say, we restrict the sorts to the empty set, thus restricting the function symbols to be empty set, and predicate symbols to propositional symbols. Is this fragment decidable? With function symbols and sorts restricted to empty set, the formulas possible are just propositional symbols and their Boolean combination. It is equivalent to the problem known as the *propositional satisfiability problem*.

The propositional satisfiability problem, also sometimes referred in computer science literature as just satisfiability problem or SAT, is one of the classical NP-complete

*AMS, 15 April 1936.

†Proceedings of the London Mathematical Society, series 2, volume 42 (1936-7); corrected version volume 43 (1937) pp. 544-546.

problems. NP-complete problems do not have any known polynomial time procedures[‡]. Despite the worst-case exponential time, modern SAT solvers can deal with very large instances of problems of interest in the real-world. But didn't we just say that no efficient algorithm is possible? Often, when the problem of interest are translated to SAT (even those that are NP-complete) they to do not give rise to the hardest instances. Advances in heuristics and new techniques has meant that industrial instances containing as many as hundreds of thousands of variables and millions of constraints can be successfully solved. That said, not all problems can be expressed in propositional logic (and often not as naturally even when they can be translated to propositional logic).

Returning to the first-order satisfiability problem, another possible approach is to sacrifice decidability. That is, allow for the algorithm to stop with "don't know" or to not terminate on some instances. A class of tools which take this approach, called *interactive theorem provers*, rely on user-assistance to help the heuristics within the solver towards a solution. Though more general, a disadvantage of this approach is that it is not fully automated.

Both approaches have their advantages, and use in appropriate place. In this thesis, we be focus on the former approach, although considering restrictions which are not as restrictive as in the exercise. The fragments we are interested in will be more expressive that propositional logic, but less expressive that the general first-order logic. We're interested in software and hardware verification domains, and accordingly will consider fragments of first-order logic that allow us to express constraints of this domain. But to do so we need a formal way to capture the restrictions. We do so in

[‡]Though it widely believed no polynomial time procedure is possible, no proof exists. It is one of the seven millennium problems identified by Clay Mathematics Institute with a prize of a million dollars for solving it!

the next section via *theories*, calling the restricted satisfiability problems *satisfiability modulo theories*.

Before we dive in, a note that the above two approaches are not completely independent. The solvers for general first-order satisfiability benefit from improvement in dedicated decision procedures for the restrictions. The interactive theorem provers can (and do) use satisfiability modulo theory solvers to discharge proof obligations if they fall in the restrictions (or allow the user to do so). Somewhat in the same fashion, satisfiability modulo theory solvers have greatly benefited and adopted ideas from the advances in (propositional) SAT solving.

Having (hopefully) convinced the reader of our place in the world, without further ado let us dive into the restriction we are interested in.

### 1.2.1 Satisfiability modulo theories

As just discussed, given the undecidability of first-order logic, it is interesting to consider restrictions which might entertain decidable procedures. A general way to talk about restrictions is through *theories*. One can define this it two ways.

Fix a signature $\Sigma$. Pick a set of $\Sigma$-sentences, say $\mathcal{K}$. Then one is interested in satisfiability of a formula $\phi$ only when $K$ holds. More formally, check whether there exists $\mathfrak{I}$ and $\nu$ such that $\mathfrak{I}, \nu \models \phi$ as well as for all $\phi' \in K$: $\mathfrak{I}, \emptyset \models \phi'$.

Another way to define a restriction is to fix a class of structures, say $\mathfrak{T}$, and try to check satisfiability only with respect to the structures in $\mathfrak{T}$. More formally, check whether: there exists a $\mathfrak{I} \in \mathfrak{T}$ and $\nu$ such that $\mathfrak{I}, \nu \models \phi$.

The satisfiability problem can be said to be modulo $\mathcal{K}$ in the first instance, and modulo $\mathfrak{T}$ in the second. The two notions above are inter-related (a set of $\Sigma$-sentences gives a set of structures wherein all are satisfied). We use the latter definition, calling

them *theories* and henceforth referring to the problem of satisfiability restricted to a theory as *satisfiability modulo theories*.

### 1.2.2 This thesis

Satisfiability modulo theory solvers have seen tremendous growth in performance in the last decade. There has been work on theories useful to reason about basic program constructs, leading to the development of dedicated procedures for theories of arithmetic, bit-vectors, arrays, and their combination. In this thesis, we will be exploring going beyond these theories.

In Chapter 2, we develop a new calculus for the theory of finite sets with cardinality constraints and prove its correctness. This is joint work with Clark Barrett, Andrew Reynolds, and Cesare Tinelli [4].

In Chapter 3, we provide a general purpose mechanism for deciding local theory extensions which capture a broad class of decidable theories useful for encoding constraints arising in software verification domains. This is joint work with Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies [6].

In Chapter 4, we discuss a new application of SMT solvers to synthesize commutativity conditions. The experimental results benefit from the decision procedure for sets. This is joint work with Eric Koskinen and Omer Tripp [5].

In Chapter 5, we provide experimental results related to the new decision procedures presented in Chapters 2 and 3 before concluding in Chapter 6.

# Chapter 2

# Theory of finite sets and cardinality

As the applications of SMT solvers have spread, so has the need for additional theories being supported directly by SMT solvers. This is sometimes for reasons of expressiveness – there is no way to directly express the constraints using existing theories. Even when the constraints can be reduced to existing theories, it is hoped that a dedicated procedure will reason more directly, and hence more efficiently, about the constraints. This is in addition to reasons related to user-friendliness such as duplication of effort for users in doing the encoding and translating the results back. To take an example of an existing theory supported by SMT solvers, the theory of linear integer arithmetic can be reduced to propositional satisfiability, but the performance is poor. A SAT solver does not reason about an addition or inequality at the level of arithmetic directly. One may also view it as loss of information in the encoding process.

In this chapter, we will explore the above question for fragments of set theory. The focus will be on developing a decision procedure which will work well in a modern SMT solver. We are interested in being able to reason about membership, typical operations on sets like union, intersection etc, and also constraints on their cardinali-

ties.

One reason to explore sets is that they are a common high-level data structure used in programming. Much like with arithmetic, arrays or bit-vectors this is useful to model the program construct directly. There are also high-level programming languages like SETL, and specification languages like B and Z, which are based on sets. More importantly, sets are one of the basic constructs of mathematics and come up quite naturally when trying to express properties of many systems.

## 2.1 Preliminaries

### 2.1.1 Language

We work in many-sorted first-order logic. The signature of our language is given by the following constants, functions, and predicates over the sorts Cardinality, Element, and Set:

1. Constant symbols: $0$, $1$ of Cardinality sort, and $\emptyset$ of Set sort.
2. Function symbols:
   - $+ : \mathsf{Cardinality} \times \mathsf{Cardinality} \to \mathsf{Cardinality}$.
   - $- : \mathsf{Cardinality} \times \mathsf{Cardinality} \to \mathsf{Cardinality}$.
   - $\sqcup : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$.
   - $\sqcap : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$.
   - $\setminus : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$.
   - $\{\cdot\} : \mathsf{Element} \to \mathsf{Set}$.
   - $|\cdot| : \mathsf{Set} \to \mathsf{Cardinality}$.

3. Predicate sybols:

   - $\approx_{\mathsf{Cardinality}}$: Cardinality $\times$ Cardinality.
   - $<$ : Cardinality $\times$ Cardinality.
   - $\approx_{\mathsf{Set}}$: Set $\times$ Set.
   - $\sqsubseteq$: Set $\times$ Set
   - $\approx_{\mathsf{Element}}$: Element $\times$ Element.
   - $\sqsubseteq$: Element $\times$ Set

We drop the subscript from $\approx$ when clear from the context.

We define the theory of finite sets with cardinality, denoted $\mathfrak{T}_S$ as the class of structures $\mathfrak{I}$, which satisfy the following properties:

1. Domains:

   - $\mathsf{Cardinality}^{\mathfrak{I}}$ must be $\mathbb{Z}$.
   - $\mathsf{Element}^{\mathfrak{I}}$ is an infinite set disjoint from $\mathsf{Cardinality}^{\mathfrak{I}}$.
   - $\mathsf{Set}^{\mathfrak{I}}$ must be all finite sets in the power set of $\mathsf{Element}^{\mathfrak{I}}$.

2. Constants are mapped to their intuitive meaning: $0^{\mathfrak{I}} = 0$, $1^{\mathfrak{I}} = 1$, $\emptyset^{\mathfrak{I}} = \emptyset$.

3. Functions are interpreted respecting their intuitive meaning:

   - $(t_1 + t_2)^{\mathfrak{I}} = t_1^{\mathfrak{I}} + t_2^{\mathfrak{I}}$.
   - $(t_1 - t_2)^{\mathfrak{I}} = t_1^{\mathfrak{I}} - t_2^{\mathfrak{I}}$.
   - $(t_1 \sqcup t_2)^{\mathfrak{I}} = t_1^{\mathfrak{I}} \cup t_2^{\mathfrak{I}}$.
   - $(t_1 \sqcap t_2)^{\mathfrak{I}} = t_1^{\mathfrak{I}} \cap t_2^{\mathfrak{I}}$.

- $(t_1 \setminus t_2)^{\mathfrak{I}} = t_1^{\mathfrak{I}} \setminus t_2^{\mathfrak{I}}$.
- $(\{t\})^{\mathfrak{I}} = \left\{t^{\mathfrak{I}}\right\}$.
- $(|t|)^{\mathfrak{I}} = \left|t^{\mathfrak{I}}\right|$.

4. Similarly, predicates are also interpreted respecting their intuitive meaning:
   - $(t_1 \approx t_2)^{\mathfrak{I}} = \mathsf{true}$ if and only if $t_1^{\mathfrak{I}} = t_2^{\mathfrak{I}}$.
   - $(t_1 < t_2)^{\mathfrak{I}} = \mathsf{true}$ if and only if $t_1^{\mathfrak{I}} < t_2^{\mathfrak{I}}$.
   - $(t_1 \sqsubseteq t_2)^{\mathfrak{I}} = \mathsf{true}$ if and only if $t_1^{\mathfrak{I}} \subseteq t_2^{\mathfrak{I}}$.
   - $(t_1 \sqsubseteq t_2)^{\mathfrak{I}} = \mathsf{true}$ if and only if $t_1^{\mathfrak{I}} \in t_2^{\mathfrak{I}}$.

For the sub-language $\mathcal{L}_A = (\{+, -\}, \{\approx_{\mathsf{Cardinality}}, <\}, \{0, 1\})$, let $\mathfrak{T}_A$ denote the set of structures which satisfy the above constraints. We assume a procedure which can check satisfiability of a formula in this language modulo $\mathfrak{T}_A$.

In this chapter, as we will only be talking about satisfiability of quantifier-free formulas, for simplifying presentation we make a technical change. For notational convenience, instead of free varialbles in the formulas, we think of them as constant symbols in an extended signature. The theories are also expanded to inlcude structures with respect to the extended signatures. Thus, instead of referring satisfiability problem being with respect to a structure in the theory and a variable assignment, we refer to the structure interepreting the additional constant symbols (we may sometimes call them variables but they are technically to be understood as constant symbols in the extended signature).

### 2.1.2 Tableau

We will describe our decision procedure as a tableau-style calculus.

### Derivation Rules

The rules of the calculus will be described as modifying a *state*:

**Definition 2.1** (State)**.** *A* state *is either the symbol* unsat *or a tuple* $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$.

Then, a rule is described as:

**Definition 2.2** (Derivation rule)**.** *A* derivation rule *is syntactically of one the following forms:*

$$\frac{P_1 \qquad P_2 \qquad P_3}{U_1 \qquad U_2}$$

$$\frac{P_1 \qquad P_2 \qquad P_3}{\textsf{unsat}}$$

*where each* $P_i$ *is a proposition over the set of states* $\sigma$*, and each* $U_i$ *is a set of updates. An update is a set of equations, where each equation is of the form* $\sigma_j := \sigma'_j$ *for some* $j \in \{1, \ldots, n\}$.

We refer to the propositions of a rule as the *premises* of the rule, and each set of updates as a *conclusion* of the rule. Rules with two or more conclusions are non-deterministic branching rules.

A rule is *applicable at state* $\sigma$ if the premise of the rule is satisfied, and if applying the updates in each of the conclusions results in a state that is different from $\sigma$.

### Derivation Trees and Derivations

Because of non-deterministic branching rules, application of the rules gives a derivation tree.

**Definition 2.3** (Derivation tree)**.** *Given a set of derivation rules $\mathcal{R}$ and a state $\sigma$, a* derivation tree with initial state $\sigma$ *is a tree where:*

- *the root node is the state $\sigma$, and*
- *for any non-leaf node $\sigma'$, there is a rule $r \in \mathcal{R}$ such that $r$ is applicable at $\sigma'$ and the children of $\sigma'$ are the result of applying the conclusions to $\sigma$ (or $\sigma'$ has a single child* unsat *if the rule's conclusion is* unsat*).*

**Definition 2.4** (Closed derivation tree)**.** *A derivation tree is said to be* closed *if the tree is finite and each branch ends in the configuration* unsat*.*

**Definition 2.5** (Open derivation tree)**.** *A derivation tree is said to be* open *if is not closed.*

**Definition 2.6** (Saturated derivation tree)**.** *Given a set of derivation rules $\mathcal{R}$, a derivation tree is said to be* saturated with respect to $\mathcal{R}$ *if for all rules $r \in \mathcal{R}$ and all leaf nodes $\sigma$ in the derivation tree, $r$ is not applicable at $\sigma$.*

**Definition 2.7** (Derives)**.** *Given a set of derivation rules $\mathcal{R}$, a derivation tree $T_1$* derives from $T_0$ *if $T_1$ is obtained from $T_0$ by the application of a single rule $r \in \mathcal{R}$ to a leaf node $\sigma$ in $T_0$. By application of a rule $r$, we mean adding as children to the leaf node the states obtained by applying the updates in each conclusion of $r$ to $\sigma$ (or a single child* unsat *if the rule's conclusion is* unsat*).*

**Definition 2.8** (Derivation)**.** *Given a set of derivation rules $\mathcal{R}$ and a state $\sigma$, a* derivation starting at $\sigma$ *is a possibly infinite sequence of derivation trees $T_0, T_1, T_2, \ldots$ where:*

- *$T_0$ is a tree with just the state $\sigma$, and*
- *for $i \geq 0$, $T_{i+1}$ derives from $T_i$.*

### 2.1.3 Graphs

A directed graph $\mathcal{G}$ is a tuple $(V(\mathcal{G}), E(\mathcal{G}))$ with $E(\mathcal{G}) \subseteq V(\mathcal{G}) \times V(\mathcal{G})$. We call $V(\mathcal{G})$ the *vertices* (or *nodes*) of the graph, and $E(\mathcal{G})$ the *edges* of the graph. A graph $\mathcal{G}$ is *acyclic* if there do not exist vertices $v_0, v_1, \ldots, v_{n-1}, v_n = v_0$ such that $(v_{i-1}, v_i) \in E(\mathcal{G})$ for all $i \in \{1 \ldots n\}$. In this work, we only work with directed acyclic graphs (or *DAG*s). For a DAG, we define the *children of a vertex* $v$, denoted $C(v)$, to be the set of vertices which have an outgoing edge from $v$. Formally, it is the set $\{w \in V(\mathcal{G}) \mid (v, w) \in E(\mathcal{G})\}$. A node $w$ is *reachable* from $v$ if either $w = v$ or there exists a node $v'$ such that $(v, v') \in E(\mathcal{G})$ and $w$ is reachable from $v'$. We call a vertex $v$ with no outgoing edges a *leaf*. In other words, $C(v) = \emptyset$ for a leaf. For a DAG, we define the *leaves of a node*, denoted $\mathrm{Leaves}\,(v)$, to be set of leaf nodes reachable from $v$. More precisely, $\mathrm{Leaves}\,(v) = \{w \in V(\mathcal{G}) \mid C(w) = \emptyset, w \text{ is reachable from } v\}$.

### 2.1.4 Notational convention

Wherever possible, we try to use following conventions. We use $x$, $y$ for variables of Element sort; $S$, $T$, $U$ for variables of Set sort; $s$, $t$, $u$ for terms of Set sort; $c$ with subscripts for variables of Cardinality sort.

## 2.2 Calculus

In this section, we describe our algorithm as a set of *derivation rules*. In the next section, we will prove the correctness of the algorithm.

**Normal form.** Given an input formula, we introduce additional variables and re-write the constraints so that the following hold:

- Each constraint is of the following form:

  1. $S \approx T$, $S \not\approx T$
  2. $S \approx \emptyset$
  3. $S \approx \{x\}$
  4. $S \approx T \sqcup U$
  5. $S \approx T \sqcap U$
  6. $S \approx T \setminus U$
  7. $x \sqsubseteq S$, $x \not\sqsubseteq S$
  8. $c_S \approx |S|$
  9. $x \approx y$, $x \not\approx y$
  10. Constraints in language $\mathcal{L}_A$ (in particular, they do not involve $|S|$ terms and use corresponding $c_S$ variables instead).

  where $S$, $T$ and $U$ are variables of Set sort, $c_S$ is a variable of Cardinality sort, and $x$, $y$ are variables of Element sort. This can be done by (i) introducing a new Set-variable for each non-variable Set-term, and adding a corresponding equality; (ii) introducing a new Cardinality-variable $c_S$ for each occurrence of $|S|$ and corresponding equality (iii) a subset constrains $S \sqsubseteq T$ is equivalent to $S \approx S \sqcup T$.

- Any set variable term appears in at most one union, intersection or set difference term. This can be done by replacing multiple occurences of the variable with new variables, and adding equality of difference variables.

- We assume an ordering on set variables. We allow only $S \sqcup T$ and $S \sqcap T$ where $T$ appears after $S$ in that ordering.

Let *set constraints*, denoted $\mathcal{S}_0$, be all constraints of the form 1-8. Let *element constraints*, denoted $\mathcal{M}_0$, be constraints of form 9. Let *arithmetic constraints*, denoted $\mathcal{A}_0$, be constraints of form 10.

**State.** As mentioned in Section 2.1.2, the rules operate on a state. In our case, the state will be denoted $\langle \mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{G} \rangle$, where $\mathcal{S}$ will be constraints involving Set terms, $\mathcal{M}$ will be equality and disequality over elements, $\mathcal{A}$ will be constraints in the sublanguage $\mathcal{L}_A$, and $\mathcal{G}$ will be a directed graph. The meaning of the first three should be clear, and we will elaborate on the graph in Section 2.2.2.

The rules and inconsistent set will be such that the tableau generated from an initial state $\langle \mathcal{S}_0, \mathcal{M}_0, \mathcal{A}_0, (\{\}, \{\}) \rangle$ will be closed if and only if the constraints are unsatisfiable.

**High-level overview.** Broadly, the rules we will describe can be divided into three categories. First are those which focus on reasoning about membership predicates (i.e. those of form $t_1 \sqsubseteq t_2$). These rules only update $\mathcal{S}$ and $\mathcal{M}$, though the premise of the rules will depend on other information in the state (in particular they depend on the vertices of the graph). Second are rules to handle constraints about the cardinality operator (i.e. those of the form $c_S \approx |S|$). The graph that we build will be central to satisfying these constraints. The final set of constraints can be thought of as exchanging additional information between the two to ensure we can combine the two models – one for cardinalities of sets, and the other specifying the elements of a set.

### 2.2.1 Set reasoning rules

The rules in this section are based on rules in Cantone and Zarba 1998 for the MLSS (multi-level syllogistic with singleton) fragment, though with some differences. First,

instead of working over just $\mathcal{S}$, the rule work over an arbitrary set of Set-terms which include all Set-terms in $\mathcal{S}$. This generalization is required to handle the interaction with cardinalities. Second, the reasoning is done modulo equality. Finally, a technical difference is that we work with ur-elements rather than untyped sets.

We introduce notation for reasoning modulo equalities. We use $\mathrm{Terms}_{\mathsf{Sort}}(\mathcal{C})$ to refer to terms of Sort sort in $\mathcal{C}$. We use $\mathrm{Terms}(\mathcal{C})$ to refer to all terms in $\mathcal{C}$. Given a set of constraints $\mathcal{C}$, we define the binary relation $\approx^*_{\mathcal{C}} \subseteq \mathcal{T} \times \mathrm{Terms}(\mathcal{C})$ to be the reflexive, symmetric, and transitive closure of the relation on terms induced by equality constraints in $\mathcal{C}$. Given this notation we define closure of member constraints $\mathcal{M}^*$ and set constraints $\mathcal{S}^*$ with respect to equality:

$$\begin{aligned}
\mathcal{M}^* = &\{x \approx y \mid x \approx^*_{\mathcal{M}} y\} \\
&\cup \{x \not\approx y \mid \exists x', y'.\ x \approx^*_{\mathcal{M}} x',\ y \approx^*_{\mathcal{M}} y',\ x' \not\approx y' \in \mathcal{M}\} \\
\mathcal{S}^* = &\mathcal{S} \cup \{x \sqsubset s \mid \exists x', s'.\ x \approx^*_{\mathcal{M}} x',\ s \approx^*_{\mathcal{S}} s',\ x' \sqsubset s' \in \mathcal{S}\} \\
&\cup \{x \not\sqsubset s \mid \exists x', s'.\ x \approx^*_{\mathcal{M}} x',\ s \approx^*_{\mathcal{S}} s',\ x' \not\sqsubset s' \in \mathcal{S}\}
\end{aligned}$$

where $x$, $y$, $x'$, $y'$ in $\mathrm{Terms}_{\mathsf{Element}}(\mathcal{M} \cup \mathcal{S})$, and $s$, $s'$ in $\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S})$. Next, we define a left-associative operator $\triangleleft$. Intuitively, $\mathcal{C} \triangleleft (P)$ updates the constraints $\mathcal{C}$ only if $P$ is not in the closure:

$$\mathcal{C} \triangleleft (P) = \begin{cases} \mathcal{C} & \text{if } P \in \mathcal{C}^* \\ \mathcal{C} \cup \{P\} & \text{otherwise} \end{cases} \tag{2.1}$$

The terms that the rules will work over will be denoted by $\mathcal{T}$. In addition to the terms from the set constraints, these will include the vertices of the graph, which are themselves Set-terms:

$$\mathcal{T} = \mathrm{Terms}(\mathcal{S}) \cup V(\mathcal{G}) \tag{2.2}$$

## Intersection

**Rule 1** (INTERSECTION DOWN 1)

$$\frac{x \sqsubseteq s \sqcap t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq s) \lhd (x \sqsubseteq t)}$$

**Rule 2** (INTERSECTION DOWN 2)

$$\frac{x \not\sqsubseteq s \sqcap t \in \mathcal{S}^* \qquad x \sqsubseteq s \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq t)}$$

**Rule 3** (INTERSECTION DOWN 3)

$$\frac{x \not\sqsubseteq s \sqcap t \in \mathcal{S}^* \qquad x \sqsubseteq t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq s)}$$

**Rule 4** (INTERSECTION UP 1)

$$\frac{x \sqsubseteq s \in \mathcal{S}^* \qquad x \sqsubseteq t \in \mathcal{S}^* \qquad s \sqcap t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq s \sqcap t)}$$

**Rule 5** (INTERSECTION UP 2)

$$\frac{x \not\sqsubseteq s \in \mathcal{S}^* \qquad s \sqcap t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq s \sqcap t)}$$

**Rule 6** (INTERSECTION UP 3)

$$\frac{x \not\sqsubseteq t \in \mathcal{S}^* \qquad s \sqcap t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq s \sqcap t)}$$

**Rule 7** (INTERSECTION SPLIT)

$$\frac{s \sqcap t \in \mathcal{T} \qquad x \sqsubseteq s \in \mathcal{S}^* \qquad x \sqsubseteq t \notin \mathcal{S}^* \qquad x \not\sqsubseteq t \notin \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq t) \qquad \mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq t)}$$

## Union

**Rule 8** (UNION DOWN 1)

$$\frac{x \notin s \sqcup t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \triangleleft (x \notin s) \triangleleft (x \notin t)}$$

**Rule 9** (UNION DOWN 2)

$$\frac{x \sqsubseteq s \sqcup t \in \mathcal{S}^* \qquad x \notin s \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq t)}$$

**Rule 10** (UNION DOWN 3)

$$\frac{x \sqsubseteq s \sqcup t \in \mathcal{S}^* \qquad x \notin t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq s)}$$

**Rule 11** (UNION UP 1)

$$\frac{x \notin s \in \mathcal{S}^* \qquad x \notin t \in \mathcal{S}^* \qquad s \sqcup t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \triangleleft (x \notin s \sqcup t)}$$

**Rule 12** (UNION UP 2)

$$\frac{x \sqsubseteq s \in \mathcal{S}^* \qquad s \sqcup t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq s \sqcup t)}$$

**Rule 13** (UNION UP 3)

$$\frac{x \sqsubseteq t \in \mathcal{S}^* \qquad s \sqcup t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq s \sqcup t)}$$

**Rule 14** (UNION SPLIT)

$$\frac{x \sqsubseteq s \sqcup t \in \mathcal{S} \qquad x \sqsubseteq s \notin \mathcal{S}^* \qquad x \sqsubseteq t \notin \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq s) \qquad \mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq t)}$$

### Set difference

**Rule 15** (Set difference Down 1)

$$\frac{x \sqsubseteq s \setminus t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq s) \lhd (x \not\sqsubseteq t)}$$

**Rule 16** (Set difference Down 2)

$$\frac{x \not\sqsubseteq s \setminus t \in \mathcal{S}^* \qquad x \sqsubseteq s \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq t)}$$

**Rule 17** (Set difference Down 3)

$$\frac{x \not\sqsubseteq s \setminus t \in \mathcal{S}^* \qquad x \not\sqsubseteq t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq s)}$$

**Rule 18** (Set difference Up 1)

$$\frac{x \sqsubseteq s \in \mathcal{S}^* \qquad x \not\sqsubseteq t \in \mathcal{S}^* \qquad s \setminus t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq s \setminus t)}$$

**Rule 19** (Set difference Up 2)

$$\frac{x \not\sqsubseteq s \in \mathcal{S}^* \qquad s \setminus t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq s \setminus t)}$$

**Rule 20** (Set difference Up 3)

$$\frac{x \sqsubseteq t \in \mathcal{S}^* \qquad s \setminus t \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq s \setminus t)}$$

**Rule 21** (Set difference split)

$$\frac{s \setminus t \in \mathcal{T} \qquad x \sqsubseteq s \in \mathcal{S}^* \qquad x \sqsubseteq t \notin \mathcal{S}^* \qquad x \not\sqsubseteq t \notin \mathcal{S}^*}{\mathcal{S} := \mathcal{S} \lhd (x \sqsubseteq t) \qquad \mathcal{S} := \mathcal{S} \lhd (x \not\sqsubseteq t)}$$

### Singleton

**Rule 22** (SINGLETON)

$$\frac{\{x\} \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \triangleleft (x \sqsubseteq \{x\})}$$

**Rule 23** (SINGLETON MEMBER)

$$\frac{x \sqsubseteq \{y\} \in \mathcal{S}^*}{\mathcal{M} := \mathcal{M} \triangleleft (x \approx y)}$$

**Rule 24** (SINGLETON NON-MEMBER)

$$\frac{x \not\sqsubseteq \{y\} \in \mathcal{S}^*}{\mathcal{M} := \mathcal{M} \triangleleft (x \not\approx y)}$$

### Disequality

Let $y$ in the following rule be a new variable of Element sort.

**Rule 25** (SET DISEQUALITY)

$$\frac{s \not\approx t \in \mathcal{S}^* \qquad \nexists x \in \mathrm{Terms}(\mathcal{S}) \text{ such that } x \sqsubseteq s \in \mathcal{S}^* \text{ and } x \not\sqsubseteq t \in \mathcal{S}^* \qquad \nexists x \in \mathrm{Terms}(\mathcal{S}) \text{ such that } x \not\sqsubseteq s \in \mathcal{S}^* \text{ and } x \sqsubseteq t \in \mathcal{S}^*}{\mathcal{S} := \mathcal{S}, y \sqsubseteq s, y \not\sqsubseteq t \qquad \mathcal{S} := \mathcal{S}, y \not\sqsubseteq s, y \sqsubseteq t}$$

### Rules (contradiction)

**Rule 26** (ELEMENT EQUALITY CONTRADICTION)

$$\frac{(x \not\approx x) \in \mathcal{M}^*}{\text{unsat}}$$

**Rule 27** (SET MEMBERSHIP CONTRADICTION)

$$\frac{(x \sqsubseteq s) \in \mathcal{S}^* \qquad (x \not\sqsubseteq s) \in \mathcal{S}^*}{\text{unsat}}$$

**Rule 28** (EMPTY SET CONTRADICTION)

$$\frac{(x \sqsubseteq \emptyset) \in \mathcal{S}^*}{\text{unsat}}$$

**Optional propagation rules**

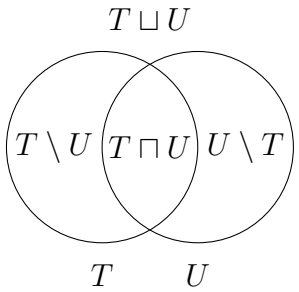
**Rule 29** (SINGLETON AND UNION)

$$\frac{x \sqsubseteq S \in \mathcal{S}^* \qquad T \approx \{x\} \in \mathcal{S}^* \qquad S \sqcup T \in \mathcal{T}}{\mathcal{S} := \mathcal{S} \triangleleft (S \approx S \sqcup T)}$$

### 2.2.2 Cardinality of sets

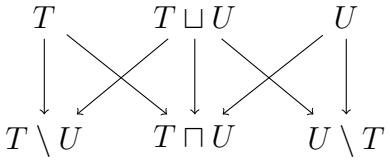
There are two forms of consistencies about the cardinality of sets that the rules in this section will handle. First, the cardinality of two sets, and their union, intersection or set difference are inter-related. They implicitly impose some constraints on cardinality of sets. Second, if two set terms are equal, their cardinalities (and also the models we build eventually) must match. These, along with consistency with respect to arithmetic constraints, and those imposed by membership constraints (handled by rules in the Section 2.2.3) will be sufficient for ensuring completeness.

Consider two sets $T$ and $U$, and how the different regions interact with each other.




As we can see union, intersection, set difference of two sets and the sets themselves affect the models and cardinalities of each other. This information is what the graph $\mathcal{G}$ will capture. The nodes of the graph will be set terms, and the children of a node

correspond to its disjoint subsets. In the example above of two sets $T$ and $U$, the interaction is captured by the graph below:




The information associated with this graph is that

- $T$ is a disjoint union of $T \setminus U$ and $T \sqcap U$;
- $T \sqcup U$ is a disjoint union of $T \setminus U$, $T \sqcap U$ and $U \setminus T$; and
- $U$ is a disjoint union of $T \sqcap U$ and $U \setminus T$.

Knowing that the sets are *disjoint* is important. It allow us to add the constraints:

$$|T| \approx |T \setminus U| + |T \sqcap U|$$
$$|T \sqcup U| \approx |T \setminus U| + |T \sqcap U| + |U \setminus T|$$
$$|U| \approx |U \setminus T| + |T \sqcap U|$$

and for any values $T \setminus U$, $T \sqcap U$ and $U \setminus T$ build a model for $T$, $U$, and $T \sqcap U$ such that everything is consistent.

This forms the basis of the rules in our calculus for cardinality. The graph should eventually contain all nodes whose cardinality is implicitly or explicitly constrained. We start by adding sets to the graph about which there are explicit constraints. Next, we add the terms whose cardinality is implicitly constrained. As discussed in the introduction to this section, these are: (i) terms which occur in an equality where the other side is in the graph, and (ii) union, intersection and set difference of terms about which there are constraints in $\mathcal{S}$, and one of the sets is in the graph. That said, a

careful analysis shows that we can avoid adding intersection terms unless both sets are in the graph, and set difference $T \setminus U$ unless $T$ is in the graph (see proof of Proposition 2.9 for why this is sufficient).

**Rule 30** (INTRODUCE CARD)

$$\frac{c_s \approx |S| \in \mathcal{S}}{\mathcal{G} := \mathrm{add}(\mathcal{G}, S)}$$

**Rule 31** (INTRODUCE EQUALITY RIGHT)

$$\frac{S \approx t \in \mathcal{S} \qquad S \in V(\mathcal{G}) \qquad t \notin V(\mathcal{G})}{\mathcal{G} := \mathrm{add}(\mathcal{G}, t)}$$

**Rule 32** (INTRODUCE EQUALITY LEFT)

$$\frac{S \approx t \in \mathcal{S} \qquad S \notin V(\mathcal{G}) \qquad t \in V(\mathcal{G})}{\mathcal{G} := \mathrm{add}(\mathcal{G}, S)}$$

**Rule 33** (INTRODUCE UNION)

$$\frac{S \approx T \sqcup U \in \mathcal{S} \qquad T \in V(\mathcal{G}) \text{ or } U \in V(\mathcal{G}) \qquad T \sqcup U \notin V(\mathcal{G})}{\mathcal{G} := \mathrm{add}(\mathcal{G}, T \sqcup U)}$$

**Rule 34** (INTRODUCE INTERSECTION)

$$\frac{S \approx T \sqcap U \in \mathcal{S} \qquad T \in V(\mathcal{G}) \qquad U \in V(\mathcal{G}) \qquad T \sqcap U \notin V(\mathcal{G})}{\mathcal{G} := \mathrm{add}(\mathcal{G}, T \sqcap U)}$$

**Rule 35** (INTRODUCE SET DIFFERENCE)

$$\frac{S \approx T \setminus U \in \mathcal{S} \qquad T \in V(\mathcal{G}) \qquad T \setminus U \notin V(\mathcal{G})}{\mathcal{G} := \mathrm{add}(\mathcal{G}, T \setminus U)}$$

where $\mathrm{add}(\mathcal{G}, s)$ is formally defined as follows:

1. For $s = T$ or $s = \emptyset$ or $s = \{x\}$, simply add $s$ to $V(\mathcal{G})$.

2. For $s = T \sqcup U$, add the following nodes to $V(\mathcal{G})$: $T$, $T \sqcup U$, $U$, $T \setminus U$, $T \sqcap U$, and $U \setminus T$. Also add the following edges to $E(\mathcal{G})$:

   - for $T$: $(T, T \setminus U)$, $(T, T \sqcap U)$,
   - for $T \sqcup U$: $(T \sqcup U, T \setminus U)$, $(T \sqcup U, T \sqcap U)$, $(T \sqcup U, U \setminus T)$,
   - for $U$: $(U, T \sqcap U)$, $(U, U \setminus T)$.

3. For $T \sqcap U$, add the following nodes to $V(\mathcal{G})$: $T$, $U$, $T \setminus U$, $T \sqcap U$, and $U \setminus T$. Also add the following edges to $E(\mathcal{G})$:

   - for $T$: $(T, T \setminus U)$, $(T, T \sqcap U)$,
   - for $U$: $(U, T \sqcap U)$, $(U, U \setminus T)$.

4. For $T \setminus U$, the case is the same as for $T \sqcap U$.

Note that, by assumption, each Set-variable participates in at most one union, intersection, or set difference. Thus, each set variable participates in at most one $\mathrm{add}$ operation. This ensures that the children of any node in the graph represent disjoint sets at introduction (this will be made more precise in Proposition 2.11).

We also need to be careful about terms which implicitly impose constraint on the cardinality.

**Rule 36** (INTRODUCE SINGLETON)

$$\frac{\{x\} \in \mathrm{Terms}(\mathcal{S})}{\mathcal{G} := \mathrm{add}(\mathcal{G}, \{x\})}$$

**Rule 37** (INTRODUCE EMPTY SET)

$$\frac{}{\mathcal{G} := \mathrm{add}(\mathcal{G}, \emptyset)}$$

Let $\mathcal{L}(n)$ denote the set of leaf nodes for the subtree rooted at node $n$ which are not known to be empty. Formally,

$$\mathcal{L}(n) = \{n' \in \text{Leaves}\,(n) \mid n' \approx \emptyset \notin \mathcal{S}^*\} \tag{2.3}$$

We call two nodes $n$ and $n'$ *merged* if they have the same set of nonempty leaves, that is if $\mathcal{L}(n) = \mathcal{L}(n')$.

The next few rules will ensure that for all equalities over set terms, the corresponding nodes in the graph are merged. Let $s \approx t \in \mathcal{S}^*$ with $s \in V(\mathcal{G})$ and $t \in V(\mathcal{G})$. Let $L_1 = \mathcal{L}(s)$ and $L_2 = \mathcal{L}(t)$. If $L_1 \subseteq L_2$ then the merge operation propagates that the nodes in $L_2 \setminus L_1$ must be empty and doesn't modify the graph (similarly for $L_2 \subseteq L_1$).

**Rule 38** (MERGE EQUALITY 1)

$$\frac{s \approx t \in \mathcal{S} \qquad s, t, \emptyset \in V(\mathcal{G}) \qquad \mathcal{L}(s) \subsetneq \mathcal{L}(t)}{\mathcal{S} := \{u \approx \emptyset \mid u \in \mathcal{L}(t) \setminus \mathcal{L}(s)\} \cup \mathcal{S}}$$

**Rule 39** (MERGE EQUALITY 2)

$$\frac{s \approx t \in \mathcal{S} \qquad s, t, \emptyset \in V(\mathcal{G}) \qquad \mathcal{L}(t) \subsetneq \mathcal{L}(s)}{\mathcal{S} := \{u \approx \emptyset \mid u \in \mathcal{L}(s) \setminus \mathcal{L}(t)\} \cup \mathcal{S}}$$

Otherwise, let $L_3 = L_1 \setminus L_2$ and $L_4 = L_2 \setminus L_1$. Add the following set of nodes and edges to the graph. We add the nodes

$$\{l_1 \sqcap l_2 \mid l_1 \in L_3, l_2 \in L_4\}$$

and for each new node $l_1 \sqcap l_2$ add the edges $(l_1, l_1 \sqcap l_2)$ and $(l_2, l_1 \sqcap l_2)$. Denote this modification of the graph by $\text{merge}(\mathcal{G}, s, t)$.

**Rule 40** (MERGE EQUALITY 3)

$$\frac{s \approx t \in \mathcal{S} \qquad s,t \in V(\mathcal{G}) \qquad \mathcal{L}(s) \not\subseteq \mathcal{L}(t) \qquad \mathcal{L}(t) \not\subseteq \mathcal{L}(s)}{\mathcal{G} := \mathrm{merge}(\mathcal{G}, s, t)}$$

**Induced arithmetic constraints.** The arithmetic constraints imposed by graph $\mathcal{G}$ are denoted by $\hat{\mathcal{G}}$. They are:

- For each set term $s \in V(\mathcal{G})$, the cardinality of the set term is the sum of the corresponding non-empty leaf nodes. Formally, the constraints are:

$$\left\{ c_s \approx \sum_{t \in \mathcal{L}(s)} c_t \;\middle|\; s \in V(\mathcal{G}) \right\}$$

- Each cardinality term should be non-negative:

$$\{c_s >= 0 \mid s \in V(\mathcal{G})\}$$

- A singleton set has cardinality 1:

$$\{c_s \approx 1 \mid s \in V(\mathcal{G}),\ s = \{x\}\}$$

- Cardinality of empty set is 0:

$$\{c_s \approx 0 \mid s \in V(\mathcal{G}),\ s = \emptyset\}$$

If the above constraints along with $\mathcal{A}$ are inconsistent, then we consider the branch of the tableau closed.

**Rule 41** (ARITHMETIC CONTRADICTION)

$$\frac{\mathcal{A} \cup \hat{\mathcal{G}} \models_{\mathfrak{T}_A} \bot}{\text{unsat}}$$

The merge rule needs to take a cross product of the leaf nodes being merged. Thus, having many sets guessed to be empty can speed up the merge operation, as well as reducing the total number of terms we need to consider significantly. We also need this rule for completeness, to be able to build a model.

**Rule 42** (GUESS EMPTY SET)

$$\frac{t \in \text{Leaves}(\mathcal{G})}{\mathcal{S} := \mathcal{S} \lhd (t \approx \emptyset) \qquad \mathcal{S} := \mathcal{S} \lhd (t \not\approx \emptyset)}$$

### 2.2.3 Cardinality and membership interaction

For this section, let $E$ denote the set of equalities in $\mathcal{M}$. By $[x]_E$ denote the equivalence class of $x$ with respect to $E$. For a Set term $t$, define $t_{\mathcal{S}}$ to be elements that must be in the set according to $\mathcal{S}$, modulo $E$. More precisely, $t_{\mathcal{S}} = \{[x]_E \mid x \sqsubseteq t \in \mathcal{S}^*\}$.

The first rule is to find an arrangement for all members in each leaf. This will be required to build a model for an open branch. This is required only if the cardinality of the set is not already lower-bounded to be at least as many elements as there are equivalence classes for elements in the set.

**Rule 43** (MEMBERS ARRANGEMENT)

$$\frac{t \in \text{Leaves}(\mathcal{G}) \qquad \mathcal{A} \not\Rrightarrow c_t \geq |t_{\mathcal{S}}| \qquad [x]_E \in t_{\mathcal{S}} \qquad [y]_E \in t_{\mathcal{S}} \qquad [x]_E \neq [y]_E \qquad x \not\approx y \notin \mathcal{M}^*}{\mathcal{M} := \mathcal{M} \lhd (x \approx y) \qquad \mathcal{M} := \mathcal{M} \lhd (x \not\approx y)}$$

where $\mathcal{A} \Rrightarrow c_t \geq |t_{\mathcal{S}}|$ holds if for some fixed $n \geq |t_{\mathcal{S}}|$, $c_t$ >= $n \in \mathcal{A}$.

**Rule 44** (PROPAGATE MINSIZE)

$$\frac{x_1 \sqsubseteq s, x_2 \sqsubseteq s, \ldots, x_n \sqsubseteq s \in \mathcal{S}^* \qquad x_i \not\approx x_j \in \mathcal{M}^* \text{ for all } 1 \leq i \neq j \leq n \qquad \mathcal{A} \not\Rightarrow c_s \geq n}{\mathcal{A} := c_s \texttt{>=} n, \mathcal{A}}$$

The following rule can be thought of as an optimization. It guesses a lower bound for the size of sets to avoid many applications of the MEMBERS ARRANGEMENT rule.

**Rule 45** (GUESS LOWER BOUND)

$$\frac{t \in \text{Leaves}(\mathcal{G}) \qquad \mathcal{A} \not\Rightarrow c_t \geq |t_{\mathcal{S}}| \qquad c_t \texttt{<} |t_{\mathcal{S}}| \notin \mathcal{A}}{\mathcal{A} := c_t \texttt{>=} |t_{\mathcal{S}}|, \mathcal{A} \qquad \mathcal{A} := c_t \texttt{<} |t_{\mathcal{S}}|, \mathcal{A}}$$

## 2.3 Correctness

We group the rules as follows:

- $\mathcal{R}_1$: Rules 1, 4, 12, 13, 14, 15, 18, 21, 22, 23, 25, 26, 27, and 28.
- $\mathcal{R}_2$: Rules 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, and 42.
- $\mathcal{R}_3$: Rule 43 and Rule 44.
- $\mathcal{R}_4$: optional propagation and split rules other than 45.
- $\mathcal{R}_5$: 45.

### 2.3.1 Completeness

The completeness involves proving the following:

**Proposition 2.9.** *Let $\mathcal{S}_0, \mathcal{M}_0, \mathcal{A}_0$ be normalized set, element and arithmetic constraints respectively. Let* **D** *be a derivation with respect to rules $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$ from state $\langle \mathcal{S}_0,$*

$\mathcal{M}_0$, $\mathcal{A}_0$, $(\{\}, \{\})\rangle$. *If* $\mathbf{D}$ *is finite, and the final derivation tree, say* $\mathcal{D}$*, in* $\mathbf{D}$ *is open and saturated with respect to the rules* $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$*; then there exists an interpretation* $\mathfrak{I}$ *that satisfies* $\mathcal{S}_0$*,* $\mathcal{M}_0$ *and* $\mathcal{A}_0$*.*

We develop the proof in stages, proving properties about different subset of rules. We start with a proposition about the rules in Section 2.2.1.

**Proposition 2.10.** *Let* $\mathcal{S}$ *and* $\mathcal{M}$ *be set and element constraints respectively. Let* $\mathcal{T}$ *an arbitrary set of* Set *terms which includes all* Set *terms in* $\mathcal{S}$*. Let* $\mathcal{S}$*,* $\mathcal{M}$ *and* $\mathcal{T}$ *be such that none of the rules in* $\mathcal{R}_1$ *are applicable.*

*Let* $\mathfrak{S}$ *be a structure interpreting all variables of* Element *sort in* $\mathcal{S}$ *and* $\mathcal{M}$ *satisfying the following property: for any* $x$ *and* $y$ *in* $\operatorname{Vars}(\mathcal{M}) \cup \operatorname{Vars}(\mathcal{S})$ *of* Element *sort,*

$$x^{\mathfrak{S}} = y^{\mathfrak{S}} \text{ if and only } x \approx y \in \mathcal{M}^* \quad .$$

*Let* $\mathfrak{S}$ *interpret each variable* $S$ *of* Set *sort in* $\operatorname{Vars}(\mathcal{S})$ *as:*

$$S^{\mathfrak{S}} = \left\{ x^{\mathfrak{S}} \;\middle|\; x \sqsubseteq S \in \mathcal{S}^* \right\}$$

*and any variable* $c_S$ *of* Cardinality *sort in* $\operatorname{Vars}(\mathcal{S})$ *as:*

$$c_S^{\mathfrak{S}} = \left| S^{\mathfrak{S}} \right| \quad .$$

*Then,* $\mathfrak{S}$ *satisfies the constraints* $\mathcal{S}$ *and* $\mathcal{M}$*.*

*Proof.* For any set term $s$, define

$$\operatorname{Elements}(s) = \left\{ x^{\mathfrak{S}} \;\middle|\; x \sqsubseteq s \in \mathcal{S}^* \right\}. \tag{2.4}$$

We will show by structural induction on set terms that for any set term $s \in \mathcal{T}$:

$$\text{Elements}(s) = s^{\mathfrak{S}} \tag{2.5}$$

*Case* 1 ($s$ is a variable). The definition of $\text{Elements}(s)$ is identical to that of $s^{\mathfrak{S}}$.

*Case* 2 ($s$ is $\emptyset$). Rule 28 (EMPTY SET CONTRADICTION) closes a branch if there is a constraint of the form $x \sqsubseteq \emptyset$ in $\mathcal{S}^*$. It follows that $\text{Elements}(\emptyset) = \emptyset$.

*Case* 3 ($s$ is $\{x\}$). As $s^{\mathfrak{S}} = \left\{x^{\mathfrak{S}}\right\}$, sufficient to show $\text{Elements}(s) = \left\{x^{\mathfrak{S}}\right\}$.

Since Rule 22 (SINGLETON) is not applicable, we can conclude that $x \sqsubseteq s \in \mathcal{S}^*$. It follows that $\left\{x^{\mathfrak{S}}\right\} \subseteq \text{Elements}(s)$.

The other direction, $\text{Elements}(s) \subseteq \left\{x^{\mathfrak{S}}\right\}$, follows from Rule 23 (SINGLETON MEMBER).

| | |
|---|---|
| $e \in \text{Elements}(\{x\})$ | |
| $e = y^{\mathfrak{S}}$ for some $y$ with $y \sqsubseteq \{x\} \in \mathcal{S}^*$ | (definition) |
| $y \approx x \in \mathcal{M}^*$ | Rule 23 (SINGLETON MEMBER) |
| $y^{\mathfrak{S}} = x^{\mathfrak{S}}$ | ($\mathfrak{S}$ satisfies $\mathcal{M}$) |
| $e \in \left\{x^{\mathfrak{S}}\right\}$ | ($e = y^{\mathfrak{S}}$) |

*Case* 4 ($s$ is $t \sqcap u$). We need to show $\text{Elements}(t \sqcap u) = t^{\mathfrak{S}} \cap u^{\mathfrak{S}}$.

First, we consider the direction $\text{Elements}(t \sqcap u) \subseteq t^{\mathfrak{S}} \cap u^{\mathfrak{S}}$. The proof depends

on Rule 1 (INTERSECTION DOWN 1).

$e \in \text{Elements}(t \sqcap u)$

$e = x^{\mathfrak{S}}$ for some $x$ with $x \sqsubseteq t \sqcap u \in \mathcal{S}^*$ (definition)

$x \sqsubseteq t \in \mathcal{S}^*$ and $x \sqsubseteq u \in \mathcal{S}^*$ (Rule 1)

$x^{\mathfrak{S}} \in \text{Elements}(t)$ and $x^{\mathfrak{S}} \in \text{Elements}(u)$ (definition)

$x^{\mathfrak{S}} \in t^{\mathfrak{S}}$ and $x^{\mathfrak{S}} \in u^{\mathfrak{S}}$ (induction)

$e \in t^{\mathfrak{S}} \cap u^{\mathfrak{S}}$

In the other direction, we show that $t^{\mathfrak{S}} \cap u^{\mathfrak{S}} \subseteq \text{Elements}(t \sqcap u)$. The reasoning relies on Rule 4 (INTERSECTION UP 1).

$e \in t^{\mathfrak{S}} \cap u^{\mathfrak{S}}$

$e \in t^{\mathfrak{S}}$ and $e \in u^{\mathfrak{S}}$

$e \in \text{Elements}(t)$ and $e \in \text{Elements}(u)$ (induction)

$x \sqsubseteq t \in \mathcal{S}^*$ and $y \sqsubseteq u \in \mathcal{S}^*$ with $x^{\mathfrak{I}} = y^{\mathfrak{I}} = e$ (definition)

$x \sqsubseteq t \in \mathcal{S}^*$ and $y \sqsubseteq u \in \mathcal{S}^*$ with $x \approx y \in \mathcal{M}^*$ (by construction)

$x \sqsubseteq t \in \mathcal{S}^*$ and $x \sqsubseteq u \in \mathcal{S}^*$

$x \sqsubseteq t \sqcap u \in \mathcal{S}^*$ ($t \sqcap u \in \mathcal{T}$, Rule 4)

$e \in \text{Elements}(t \sqcap u)$

*Case* 5 ($s$ is $t \sqcup u$). We need to show $\text{Elements}(t \sqcup u) = t^{\mathfrak{S}} \cup u^{\mathfrak{S}}$.

First, we consider the direction $\text{Elements}(t \sqcup u) \subseteq t^{\mathfrak{S}} \cap u^{\mathfrak{S}}$. The reasoning relies

on Rule 14 (UNION SPLIT).

$e \in \text{Elements}(t \sqcup u)$

$e = x^{\mathfrak{S}}$ for some $x$ with $x \sqsubseteq t \sqcup u \in \mathcal{S}^*$ (definition)

$x \sqsubseteq t \in \mathcal{S}^*$ or $x \sqsubseteq u \in \mathcal{S}^*$ (Rule 14 (UNION SPLIT))

$x^{\mathfrak{S}} \in \text{Elements}(t)$ or $x^{\mathfrak{S}} \in \text{Elements}(u)$ (definition)

$x^{\mathfrak{S}} \in t^{\mathfrak{S}}$ or $x^{\mathfrak{S}} \in u^{\mathfrak{S}}$ (induction)

$e \in t^{\mathfrak{S}} \cup u^{\mathfrak{S}}$

In the other direction, we show that $t^{\mathfrak{S}} \cup u^{\mathfrak{S}} \subseteq \text{Elements}(t \sqcup u)$. The reasoning follows directly from Rule 12 (UNION UP 2) and Rule 13 (UNION UP 3).

$e \in t^{\mathfrak{S}} \cup u^{\mathfrak{S}}$

$e \in t^{\mathfrak{S}}$ or $e \in u^{\mathfrak{S}}$

$e \in \text{Elements}(t)$ or $e \in \text{Elements}(u)$ (induction)

$x \sqsubseteq t \in \mathcal{S}^*$ or $x \sqsubseteq u \in \mathcal{S}^*$ where $x^{\mathfrak{I}} = e$ (definition)

$x \sqsubseteq t \sqcup u \in \mathcal{S}^*$ ($t \sqcup u \in \mathcal{T}$, Rule 12 or 13)

$e \in \text{Elements}(t \sqcap u)$

*Case 6* ($s$ is $t \setminus u$). We need to show $\text{Elements}(t \setminus u) = t^{\mathfrak{S}} \setminus u^{\mathfrak{S}}$.

First, we consider the direction $\text{Elements}(t \setminus u) \subseteq t^{\mathfrak{S}} \setminus u^{\mathfrak{S}}$. The proof depends on Rule 15 (SET DIFFERENCE DOWN 1) and Rule 27 (SET MEMBERSHIP CONTRADIC-

TION).

$$
\begin{aligned}
& e \in \text{Elements}(t \setminus u) \\
& e = x^{\mathfrak{S}} \text{ for some } x \text{ with } x \sqsubseteq t \setminus u \in \mathcal{S}^* && \text{(definition)} \\
& x \sqsubseteq t \in \mathcal{S}^* \text{ and } x \not\sqsubseteq u \in \mathcal{S}^* && \text{(Rule 15)}
\end{aligned}
$$

We show by contradiction that $x^{\mathfrak{S}} \notin \text{Elements}(u)$. If possible, let $x^{\mathfrak{S}} \in \text{Elements}(u)$. That is, there exists $y$ such that $y^{\mathfrak{S}} = x^{\mathfrak{S}}$ and $y \sqsubseteq u \in \mathcal{S}^*$. By definition of $\mathfrak{S}$, it follows that $x \approx y \in \mathcal{M}^*$. So, $x \sqsubseteq u \in \mathcal{S}^*$. Contradiction by Rule 27 (SET MEMBERSHIP CONTRADICTION). We conclude,

$$
\begin{aligned}
& x^{\mathfrak{S}} \in \text{Elements}(t) \text{ and } x^{\mathfrak{S}} \notin \text{Elements}(u) \\
& x^{\mathfrak{S}} \in t^{\mathfrak{S}} \text{ and } x^{\mathfrak{S}} \notin u^{\mathfrak{S}} && \text{(induction)} \\
& e \in t^{\mathfrak{S}} \setminus u^{\mathfrak{S}}
\end{aligned}
$$

In the other direction, we show that $t^{\mathfrak{S}} \setminus u^{\mathfrak{S}} \subseteq \text{Elements}(t \setminus u)$. The reasoning relies on Rule 18 (SET DIFFERENCE UP 1) and Rule 21 (SET DIFFERENCE SPLIT).

$$
\begin{aligned}
& e \in t^{\mathfrak{S}} \setminus u^{\mathfrak{S}} \\
& e \in t^{\mathfrak{S}} \text{ and } e \notin u^{\mathfrak{S}} \\
& e \in \text{Elements}(t) \text{ and } e \notin \text{Elements}(u) && \text{(induction)} \\
& x \sqsubseteq t \in \mathcal{S}^* \text{ and } x \sqsubseteq u \notin \mathcal{S}^* \text{ for some } x \text{ with } x^{\mathfrak{I}} = e && \text{(definition)}
\end{aligned}
$$

We show by contradiction that $x \not\sqsubseteq u \in \mathcal{S}^*$. If possible, let $x \not\sqsubseteq u \notin \mathcal{S}^*$. As $x \sqsubseteq u \notin \mathcal{S}^*$ and $t \setminus u \in \mathcal{T}$, the premise of Rule 21 (SET DIFFERENCE SPLIT) is satisfied. As we had neither $x \sqsubseteq u \in \mathcal{S}^*$ nor $x \not\sqsubseteq u \in \mathcal{S}^*$, we get a contradiction.

| | |
|---|---|
| $x \sqsubseteq t \in \mathcal{S}^*$ and $x \not\sqsubseteq u \in \mathcal{S}^*$ | (Rule 21) |
| $x \sqsubseteq t \setminus u \in \mathcal{S}^*$ | ($t \setminus u \in \mathcal{T}$, Rule 18) |
| $e \in \text{Elements}(t \setminus u)$ | |

Having established the property of $\text{Elements}(\cdot)$, showing that each constraint in $\mathcal{S}$ is satisfied by $\mathfrak{S}$ is straightforward:

1. Let $x \sqsubseteq s \in \mathcal{S}$. Then, $x^{\mathfrak{S}} \in \text{Elements}(s)$. With the property of $\text{Elements}(\cdot)$ just established it follows $x^{\mathfrak{S}} \in s^{\mathfrak{S}}$.

2. Let $x \not\sqsubseteq s \in \mathcal{S}$. We show $x^{\mathfrak{S}} \notin s^{\mathfrak{S}}$ by contradiction.

   | | |
   |---|---|
   | $x^{\mathfrak{S}} \in s^{\mathfrak{S}}$ | (assume) |
   | $x^{\mathfrak{S}} \in \text{Elements}(s)$ | (proved above) |
   | $x^{\mathfrak{S}} = y^{\mathfrak{S}}$ for some $y$ with $y \sqsubseteq s \in \mathcal{S}^*$ | (definition) |
   | $x \approx y \in \mathcal{M}^*$ | ($x^{\mathfrak{S}} = y^{\mathfrak{S}}$ iff $x \approx y \in \mathcal{M}^*$) |
   | $x \sqsubseteq s \in \mathcal{S}^*$ | (definition of $\mathcal{S}^*$) |
   | Tableau is closed, contradiction. | (Rule 27) |

3. Let $s \approx t \in \mathcal{S}$. From the definition of $\mathcal{S}^*$ it follows that $\text{Elements}(s) = \text{Elements}(t)$. Since $s^{\mathfrak{S}} = \text{Elements}(s)$ and $t^{\mathfrak{S}} = \text{Elements}(t)$, it follows that $s^{\mathfrak{S}} = t^{\mathfrak{S}}$.

4. Let $s \not\approx t \in \mathcal{S}$. From Rule 25 (SET DISEQUALITY), it follows that there exists $x$ such that either $x \sqsubset s \in \mathcal{S}^*$ and $x \not\sqsubset t \in \mathcal{S}^*$, or $x \not\sqsubset s \in \mathcal{S}^*$ and $x \sqsubset t \in \mathcal{S}^*$. It follows that either $x^{\mathfrak{S}} \in s^{\mathfrak{S}}$ and $x^{\mathfrak{S}} \notin t^{\mathfrak{S}}$, or $x^{\mathfrak{S}} \notin s^{\mathfrak{S}}$ and $x^{\mathfrak{S}} \in t^{\mathfrak{S}}$. In either case, we can conclude that $s^{\mathfrak{S}} \neq t^{\mathfrak{S}}$.

5. Let $c_S \approx |S| \in \mathcal{S}$. By definition, both $c_S^{\mathfrak{S}} = \left|S^{\mathfrak{S}}\right| = |S|^{\mathfrak{S}}$.

□

Next, about the rules in Section 2.2.2.

**Proposition 2.11.** *Let* **D** *be a derivation with respect to rules in our calculus starting from state* $\langle \mathcal{S}_0, \mathcal{M}_0, \mathcal{A}_0, (\emptyset, \emptyset)\rangle$. *Let* $\mathcal{D} \in \mathbf{D}$ *that is open and saturated. Let* $\langle \mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{G}\rangle$ *be the final state on a branch in* $\mathcal{D}$ *that is not* unsat. *Then,* $\mathcal{G}$ *satisfies the following properties:*

1. *If* $s \in V(\mathcal{G})$ *or* $t \in V(\mathcal{G})$, *and* $s \approx t \in \mathcal{S}$; *then* $\mathcal{L}(s) = \mathcal{L}(t)$.

2. *For a node* $T \sqcup U$, $\mathcal{L}(T \sqcup U) = \mathcal{L}(T) \cup \mathcal{L}(U)$.

3. *For a node* $T \sqcap U$, $\mathcal{L}(T \sqcap U) = \mathcal{L}(T) \cap \mathcal{L}(U)$.

4. *For a node* $T \setminus U$, $\mathcal{L}(T \setminus U) = \mathcal{L}(T) \setminus \mathcal{L}(U)$.

5. *Let* $s \in V(\mathcal{G})$. *For all* $t, u \in \mathrm{Leaves}\,(s)$, $t \neq u$, *the two sets are necessarily disjoint:*

$$\models_{\mathfrak{T}_S} t \sqcap u \approx \emptyset.$$

*6. Let $s \in V(\mathcal{G})$.Let $E = \{t \approx u \mid t \approx u \in \mathcal{S}^*\}$. Then,*[*]

$$\models_{\mathfrak{T}_S} \left( \bigwedge_{P \in E} P \right) \Rightarrow \left( s \approx \bigsqcup_{t \in \mathcal{L}(s)} t \right) \tag{2.6}$$

*Proof (Proposition 2.11, property 1).* Let $s \approx t \in \mathcal{S}$, with $s \in V(\mathcal{G})$ or $t \in V(\mathcal{G})$. From Rule 31 (INTRODUCE EQUALITY RIGHT) and Rule 32 (INTRODUCE EQUALITY LEFT) it follows that both $s \in V(\mathcal{G})$ and $t \in V(\mathcal{G})$. For each of the Rules 38, 39, and 40; we show that after the application of the rule, $\mathcal{L}(s)$ and $\mathcal{L}(t)$ are equal.

Consider Rule 38 (MERGE EQUALITY 1). Let $L_s$ and $L_t$ denote $\mathcal{L}(s)$ and $\mathcal{L}(t)$ respectively before application of the rule. Let $L'_s$ and $L'_t$ denote $\mathcal{L}(s)$ and $\mathcal{L}(t)$ after application of the rule. For the rule to be applicable $L_s \subsetneq L_t$. The rule adds constraints to $\mathcal{S}$ so that $L'_t = L_t \setminus (L_t \setminus L_s)$. Equivalently, $L'_t = L_t \cap L_s = L_s$. Since $L'_s = L_s$, we get $L'_s = L_s = L'_t$.

The case for Rule 39 (MERGE EQUALITY 2) is analogous to Rule 38 (MERGE EQUALITY 1).

Consider Rule 40 (MERGE EQUALITY 3). Let $L_s$ and $L_t$ denote $\mathcal{L}(s)$ and $\mathcal{L}(t)$ respectively before application of the rule. Let $L'_s$ and $L'_t$ denote $\mathcal{L}(s)$ and $\mathcal{L}(t)$ after application of the rule. Let $n \in L'_s$. Note that the $\mathrm{merge}$ operation only adds nodes and vertices. Thus, $n$ is one of the following:

- $l_1 \sqcap l_2$ with $l_1 \in L_s$ and $l_2 \in L_t$: Since $(l_1, l_1 \sqcap l_2)$ as well as $(l_2, l_1 \sqcap l_2)$ is an edge, it follows that $n \in L'_t$.

- $l_1 \in L_s$. Since nodes in $L_s \setminus L_t$ have an outgoing edge, it must be the case that $l_1 \in L_s \cap L_t$. It follows that $n \in L'_t$.

[*]Technically, $\bigsqcup_{\dots}$ is ambiguous. But, for any structure in $\mathfrak{T}_S$, the interpretation of $\sqcup$ is associative, so the bracketing doesn't matter in our context.

This shows that $L'_s \subseteq L'_t$. The reasoning for $L'_t \subseteq L'_s$ is symmetrical.

As $s \approx t$, $s \in V(\mathcal{G})$, and $t \in V(\mathcal{G})$, the premise of at least once of the rules (38), (39), and (40) must be satisfied whenever $\mathcal{L}(s) \neq \mathcal{L}(t)$. As the branch is saturated, $\mathcal{L}(s) = \mathcal{L}(t)$ follows. □

*Proof (Proposition 2.11, properties 2, 3, 4).* As $\mathcal{D}$ is obtained from a derivation starting with a state with an empty graph, it is sufficient to show the properties hold for the empty graph, and that they are preserved each time the graph is modified by one of the rules.

The properties hold trivially for the empty graph. The interesting cases are when edges are added to the graph: i) $\mathrm{add}$ of a union, intersection, or set minus term, and ii) $\mathrm{merge}$ operation.

Observe that when we introduce $T \sqcup U$, $T \sqcap U$, and $T \setminus U$ to the graph:

- $\mathrm{Leaves}\,(T) = \{T \setminus U, T \sqcap U\}$,
- $\mathrm{Leaves}\,(U) = \{T \sqcap U, U \setminus T\}$,
- $\mathrm{Leaves}\,(T \sqcup U) = \{T \setminus U, T \sqcap U, U \setminus T\}$,
- $\mathrm{Leaves}\,(T \sqcap U) = \{T \sqcap U\}$,
- $\mathrm{Leaves}\,(T \setminus U) = \{T \setminus U\}$, and
- $\mathrm{Leaves}\,(U \setminus T) = \{U \setminus T\}$.

We conclude that:

- $\mathrm{Leaves}\,(T \sqcup U) = \mathrm{Leaves}\,(T) \cup \mathrm{Leaves}\,(U)$
- $\mathrm{Leaves}\,(T \sqcap U) = \mathrm{Leaves}\,(T) \cap \mathrm{Leaves}\,(U)$

- $\text{Leaves}\,(T \setminus U) = \text{Leaves}\,(T) \setminus \text{Leaves}\,(U)$
- $\text{Leaves}\,(U \setminus T) = \text{Leaves}\,(U) \setminus \text{Leaves}\,(T)$

when an introduce rule is applied. Note that the merge operation only adds edges from existing leaf nodes, ensuring that the property is maintained by any application of merge.

$\mathcal{L}(\cdot)$, as defined in (2.3), can also be defined as:

$$\mathcal{L}(n) = \text{Leaves}\,(n) \setminus E \tag{2.7}$$

where $E = \{n' \in V(\mathcal{G}) \mid n' \approx \emptyset \in \mathcal{S}^*\}$ does not depend on $n$. The properties in the proposition about $\mathcal{L}(\cdot)$ follow from the corresponding property of $\text{Leaves}\,(\cdot)$ just established, and above formulation of $\mathcal{L}(\cdot)$. □

*Proof (Proposition 2.11, properties 5,6).* The properties holds trivially for the empty graph.

Let $\mathcal{G}$ be the graph constraints. Let $s \in V(\mathcal{G})$. Let $s' \approx \emptyset$ be a new constraint such that $s' \in \mathcal{L}(s)$. Then, this modifies $\mathcal{L}(s)$, and we need to verify the Property 6 still holds. Note that for any structure in $\mathfrak{T}_S$, if $s'$ is interpreted as empty set, the interpretation of $\bigsqcup_{t \in \mathcal{L}(s) \setminus \{s'\}} t$ will be same as $\bigsqcup_{t \in \mathcal{L}(s)} t$. Thus, if $s' \in \mathcal{L}(s)$ and

$$\models_{\mathfrak{T}_S} \left( \bigwedge_{P \in E} P \right) \Rightarrow \left( s \approx \bigsqcup_{t \in \mathcal{L}(s)} t \right) \quad ,$$

then

$$\models_{\mathfrak{T}_S} \left( s' \approx \emptyset \wedge \bigwedge_{P \in E} P \right) \Rightarrow \left( s \approx \bigsqcup_{t \in \mathcal{L}(s) \setminus \{s'\}} t \right) \quad .$$

It follows if $s' \approx \emptyset$ is added to $\mathcal{S}^*$ by a rule, the property 6 continue to hold. Also note

that a equality is not removed by any rule (if there was such a rule, we'd need to check the property continues to hold when the left side of the implication is weakened).

The only other rules which affect the properties are those which modify the graph directly, i.e. the $\mathrm{add}$ and $\mathrm{merge}$ operations.

We show that if $\mathcal{G}$ satisfies the properties, then so does $\mathrm{add}(\mathcal{G}, s)$:

- $s$ is $\emptyset$, $S$ or $\{x\}$: trivially, as no edges are added.

- $s$ is $T \sqcap U$: Note that because of the assumptions on the normal form, either $T \sqcap U$ already in the graph and $\mathrm{add}$ operation doesn't modify the graph, or it will add the nodes $T$, $U$, $T \setminus U$, $T \sqcap U$, and $U \setminus T$ to the graph, and edges between them. It is easy to see that the property 5 follows from:

$$\models_{\mathfrak{T}_S} ((T \setminus U) \sqcap (T \sqcap U)) \approx \emptyset$$
$$\models_{\mathfrak{T}_S} ((U \setminus T) \sqcap (T \sqcap U)) \approx \emptyset$$

  Property 6 follows from:

$$\models_{\mathfrak{T}_S} T \approx ((T \setminus U) \sqcup (T \sqcap U))$$
$$\models_{\mathfrak{T}_S} U \approx ((U \setminus T) \sqcup (T \sqcap U))$$
$$\models_{\mathfrak{T}_S} (T \sqcap U) \approx (T \sqcap U)$$
$$\models_{\mathfrak{T}_S} (U \setminus T) \approx (U \setminus T)$$
$$\models_{\mathfrak{T}_S} (T \setminus U) \approx (T \setminus U)$$

  and reasoning as earlier that any constraint of the form $s' \approx \emptyset$ doesn't affect the property.

- $s$ is $T \setminus U$ or $U \setminus T$: reasoning same as for $T \sqcap U$.

- $s$ is $T \sqcup U$. If not already present, $T$, $U$, $T \setminus U$, $T \sqcap U$ are added to the graph as for $T \sqcap U$. In addition, $\mathrm{add}$ for union also adds $T \sqcup U$, and three edges. The properties follows from the following tautologies in $\mathfrak{T}_S$ in addition to those listed in analysis for $T \sqcap U$:

$$\models_{\mathfrak{T}_S} ((T \setminus U) \sqcap ((U \setminus T))) \approx \emptyset$$
$$\models_{\mathfrak{T}_S} (T \sqcup U) \approx ((T \setminus U) \sqcup (T \sqcap U) \sqcup (U \setminus T))$$

Finally, we show that if $\mathcal{G}$ satisfies the properties, then so does $\mathrm{merge}(\mathcal{G}, s, t)$ if $s \in V(\mathcal{G})$, $t \in V(\mathcal{G})$, $\mathcal{L}(s) \not\subseteq \mathcal{L}(t)$ and $\mathcal{L}(t) \not\subseteq \mathcal{L}(s)$.

Let $L_s$ denote $\mathcal{L}(s)$ in $\mathcal{G}$, and $L'_s$ denote $\mathcal{L}(s)$ in $\mathrm{merge}(\mathcal{G}, s', t')$ (likewise for $t$, $u$ etc.).

In order to show property 5 holds, let $s' \in V(\mathcal{G})$, $t' \in L'_{s'}$ and $u' \in L'_{s'}$. We need to show: $\models_{\mathfrak{T}_S} t' \sqcap u' \approx \emptyset$.

- Let $t' \in L_{s'}$ and $u' \in L_{s'}$, i.e. both are also leaf nodes in $\mathcal{G}$. Then, the property for $\mathrm{merge}(\mathcal{G}, s, t)$ follows from that of $\mathcal{G}$.

- Let $t'$ be one of the newly introduced leaf nodes and $u' \in L_{s'}$ a leaf node in $\mathcal{G}$. Without loss of generality, let $t'$ be $t_1 \sqcap t_2$ with $t_1 \in L_s \setminus L_t$ and $t_2 \in L_t \setminus L_s$. For $t'$ to be in $L'_{s'}$, given the way the edges are added, either $t_1 \in L_{s'}$ or $t_2 \in L_{s'}$. Thus, we know that either $\models_{\mathfrak{T}_S} t_1 \sqcap u' \approx \emptyset$ or $\models_{\mathfrak{T}_S} t_2 \sqcap u' \approx \emptyset$. In either case, it follows that $\models_{\mathfrak{T}_S} (t_1 \sqcap t_2) \sqcap u' \approx \emptyset$, i.e. $\models_{\mathfrak{T}_S} t' \sqcap u' \approx \emptyset$.

- The analysis for the case where both are newly introduced leaf nodes is similar.

To show property 6 holds, the main observation is that each node no longer a leaf node, say $s' \in L_s \setminus L'_s$, is union of a new set of leaf nodes in $L'_s$ (assuming the equalities).

$$
\begin{aligned}
s' &\approx s' \sqcap s && (s' \in L_s, s \approx \bigsqcup_{s'' \in L_s} s'') \\
&\approx s' \sqcap t && (s \approx t \in E) \\
&\approx s' \sqcap \left( \bigsqcup_{t' \in L_t} t' \right) && (t \approx \bigsqcup_{t' \in L_t} t') \\
&\approx \bigsqcup_{t' \in L_t} s' \sqcap t' && \text{(distribute)}
\end{aligned}
$$

But by property 5, $s' \sqcap t' \approx \emptyset$ for $s', t' \in L_s$. Thus,

$$s' \approx \bigsqcup_{t' \in L_t \setminus L_s} s' \sqcap t'$$

Note that $\{s' \sqcap t' \mid t' \in L_t \setminus L_s\}$ are precisely the nodes in $L'_s$ to which edges are added from $s'$. The proof for a node in $L_t$ but not in $L'_t$ is similar.

Since all the new leaf nodes are of the form $s' \sqcap t'$ with $s' \in L_s \setminus L_t$ and $t' \in L_t \setminus L_s$, it follows that property 6 holds for $\mathrm{merge}(\mathcal{G}, s, t)$ if it holds for $\mathcal{G}$ assuming $s \approx t \in E$. □

Third, about the rules in Section 2.2.3.

**Proposition 2.12.** *Let $\langle \mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{G} \rangle$ be a state such that none of the rules in our calculus are applicable. Let $\mathfrak{S}$ be a structure defined in Proposition 2.10 satisfying constraints in $\mathcal{S}$ and $\mathcal{M}$. To recall, for $x$ and $y$ of* Element *sort,*

$$x^{\mathfrak{S}} = y^{\mathfrak{S}} \textit{ if and only if } x \approx y \in \mathcal{M}^*$$

*and for $s$ of Set sort,*

$$s^{\mathfrak{S}} = \left\{ x^{\mathfrak{S}} \;\middle|\; x \sqsubseteq s \in \mathcal{S}^* \right\}.$$

*Let $\mathfrak{A}$ be a structure satisfying $\mathcal{A}$. Then, for all $t \in \mathcal{L}(\mathcal{G})$,*

$$c_t^{\mathfrak{A}} \geq \left| t^{\mathfrak{S}} \right|.$$

*Proof.* Let $t \in \mathcal{L}(\mathcal{G})$. First we show that if $\mathcal{A} \Rightarrow c_t \geq |t_{\mathcal{S}}|$, then the proposition follows. That is there exists $n \geq |t_{\mathcal{S}}|$ such that $c_t$ >= $n \in \mathcal{A}$. Let $\mathrm{Elements}(\cdot)$ be as in (2.4).

$$\begin{aligned}
c_t^{\mathfrak{A}} &\geq n^{\mathfrak{A}} && (c_t \texttt{>=} n \in \mathcal{A}) \\
&= n && \text{(constant symbol)} \\
&\geq |t_{\mathcal{S}}| && \text{(definition)} \\
&= |\mathrm{Elements}(t)| && (x^{\mathfrak{S}} = y^{\mathfrak{S}} \text{ iff } x \approx y \in \mathcal{M}^*) \\
&= \left| t^{\mathfrak{S}} \right| && \text{(using (2.5))}
\end{aligned}$$

It remains to show that $\mathcal{A} \Rightarrow c_t \geq |t_{\mathcal{S}}|$. Because of Rule 43 (MEMBERS ARRANGEMENT), either $\mathcal{A} \Rightarrow c_t \geq |t_{\mathcal{S}}|$ or Rule 43 is applicable until the premise of Rule 44 (PROPAGATE MINSIZE) holds. If Rule 44 is applicable, $c_t$ >= $|t_{\mathcal{S}}|$ must have been added to $\mathcal{A}$. In either case, $\mathcal{A} \Rightarrow c_t \geq |t_{\mathcal{S}}|$. □

Now we return to the proof of the main result of this section, Proposition 2.9. Let us recall the proposition:

> Let $\mathcal{S}_0, \mathcal{M}_0, \mathcal{A}_0$ be normalized set, element and arithmetic constraints respectively. Let $\mathbf{D}$ be a derivation with respect to rules $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$ from

state $\langle \mathcal{S}_0, \mathcal{M}_0, \mathcal{A}_0, (\{\}, \{\}) \rangle$. If $\mathbf{D}$ is finite, and the final derivation tree, say $\mathcal{D}$, in $\mathbf{D}$ is open and saturated with respect to the rules $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$; then there exists an interpretation $\mathfrak{I}$ that satisfies $\mathcal{S}_0$, $\mathcal{M}_0$ and $\mathcal{A}_0$.

Intuitively, we start by building the model of the leaf nodes in the graphs using the model obtained from Proposition 2.10. We add additional elements to these sets to make the cardinalities match the model satisfying the arithmetic constraints and the constraints induced by the graph. Propositions 2.11 and 2.12 ensure that this is always possible to do without violating the set constraints.

*Proof of Proposition 2.9.* As $\mathcal{D}$ is open, there exists a branch that doesn't end in the state unsat. Let $\langle \mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{G} \rangle$ be the final state on such a branch.

Let $\mathcal{A} \cup \hat{\mathcal{G}}$ be the arithmetic constraints, and the arithmetic constraints induced by the graph. These constraints fall in the theory $\mathfrak{T}_A$. Let $\mathfrak{A}$ be the structure satisfying these constraints. Such an structure exists because Rule 41 (ARITHMETIC CONTRADICTION) would have closed the branch if the constraints were inconsistent. From Proposition 2.10, we obtain an structure $\mathfrak{S}$ satisfying $\mathcal{S}$ and $\mathcal{M}$. Without loss of generality, assume that $\mathsf{Element}^{\mathfrak{S}}$ is infinite.

The $\mathfrak{I}$ we build satisfying $\mathcal{S}_0 \cup \mathcal{M}_0 \cup \mathcal{A}_0$ will be as follows. It coincides with the structure $\mathfrak{S}$ on terms of Element sort. It coincides with the structure $\mathfrak{A}$ on terms of Cardinality sort. In order to define the value of Set variables, for each leaf node $t \in \text{Leaves}(\mathcal{G})$ we create the following sets:

$$B_t = \{e_{t,1}, e_{t,2} \ldots e_{t,c_t^{\mathfrak{I}} - |t^{\mathfrak{S}}|}\}$$

where $e_{t,i} \in \mathsf{Element}^{\mathfrak{S}}$ are distinct from each other and from any $e$ such that $e = x^{\mathfrak{S}}$ for $x$ in $\mathcal{S}$ or $\mathcal{M}$. From Proposition 2.12, we know that $c_t^{\mathfrak{I}} \geq \left|t^{\mathfrak{S}}\right|$. Thus, for a leaf

node $t$,

$$\left|t^{\mathfrak{S}}\right| + |B_t| = c_t^{\mathfrak{I}}. \tag{2.8}$$

For a set variable not in the graph, $S \notin V(\mathcal{G})$, define $S^{\mathfrak{I}} = S^{\mathfrak{S}}$. For a set variable in the graph, $S \in V(\mathcal{G})$, define:

$$S^{\mathfrak{I}} = \bigcup_{t \in \mathcal{L}(S)} (t^{\mathfrak{S}} \cup B_t) \tag{2.9}$$

From Proposition 2.11, it follows that:

$$\bigcup_{t \in \mathcal{L}(S)} t^{\mathfrak{S}} = S^{\mathfrak{S}} \tag{2.10}$$

So an equivalent way to define $S^{\mathfrak{I}}$ is as follows:

$$S^{\mathfrak{I}} = S^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(S)} B_t \tag{2.11}$$

We verify that each constraints in $\mathcal{S}_0$ is satisfied:

1. $S \approx T, S \not\approx T$.

   For $S \approx T$, we need to show $S^{\mathfrak{I}} = T^{\mathfrak{I}}$. If neither $S \in V(\mathcal{G})$ nor $T \in V(\mathcal{G})$, then this follows from Proposition 2.10. If either $S \in V(\mathcal{G})$ or $T \in V(\mathcal{G})$, then due to Rule 31 (INTRODUCE EQUALITY RIGHT) and Rule 32 (INTRODUCE EQUALITY LEFT) both $S \in V(\mathcal{G})$ and $T \in V(\mathcal{G})$. From Proposition 2.11, property 1, we know that $\mathcal{L}(S) = \mathcal{L}(T)$. From the definition of $S^{\mathfrak{I}}$ and $T^{\mathfrak{I}}$ in (2.9), it follows that $S^{\mathfrak{I}} = T^{\mathfrak{I}}$.

   For $S \not\approx T$, we need to show $S^{\mathfrak{I}} \neq T^{\mathfrak{I}}$. Let us write $S^{\mathfrak{I}} = S^{\mathfrak{S}} \cup B_S$, where $B_S = \emptyset$ if $S \notin V(\mathcal{G})$, otherwise let $B_S = \bigcup_{t \in \mathcal{L}(S)} B_t$ (from (2.11)). Similarly

we may write $T^{\mathfrak{I}} = T^{\mathfrak{S}} \cup B_T$. From Proposition 2.10 we know that $S^{\mathfrak{S}} \neq T^{\mathfrak{S}}$. Without loss of generality assume $e \in S^{\mathfrak{S}}$ and $e \notin T^{\mathfrak{S}}$. By definition, $B_T$ is disjoint from $S^{\mathfrak{S}}$, thus $e \notin B_T$. Thus, $e \in S^{\mathfrak{I}}$ and $e \notin T^{\mathfrak{I}}$. $S^{\mathfrak{I}} \neq T^{\mathfrak{I}}$ follows.

2. $S \approx \emptyset$.

   We need to show $S^{\mathfrak{I}} = \emptyset^{\mathfrak{I}} = \emptyset$. It will follow from Rule 37 (INTRODUCE EMPTY SET) and Rule 32 (INTRODUCE EQUALITY LEFT).

$$\begin{aligned}
&\emptyset \in V(\mathcal{G}) \text{ and } S \in V(\mathcal{G}) && \text{(Rules 37, 32)}\\
&\mathcal{L}(S) = \mathcal{L}(\emptyset) && \text{(Proposition 2.11, property 1)}\\
&\mathcal{L}(S) = \emptyset && (\mathcal{L}(\emptyset) = \emptyset)\\
&S^{\mathfrak{I}} = \emptyset && (S \in V(\mathcal{G}), (2.9))
\end{aligned}$$

3. $S \approx \{x\}$.

   We need to show that $S^{\mathfrak{I}} = \left\{x^{\mathfrak{I}}\right\}$. From Rule 36 (INTRODUCE SINGLETON) we conclude that $\{x\} \in V(\mathcal{G})$ Then, from Rule 32 (INTRODUCE EQUALITY LEFT), $S \in V(\mathcal{G})$.

   From $\hat{\mathcal{G}}$, we know that:

$$\begin{aligned}
c_S^{\mathfrak{I}} &= \sum_{t \in \mathcal{L}(S)} c_t^{\mathfrak{I}} && \text{(constraint in } \hat{\mathcal{G}} \text{ for } c_S)\\
&= \sum_{t \in \mathcal{L}(\{x\})} c_t^{\mathfrak{I}} && \text{(Proposition 2.11, property 1)}\\
&= c_{\{x\}}^{\mathfrak{I}} && \text{(constraint in } \hat{\mathcal{G}} \text{ for } c_{\{x\}})\\
&= 1 && \text{(constraint in } \hat{\mathcal{G}} \text{ for singletons)}
\end{aligned}$$

We can conclude that $\left|S^{\mathfrak{I}}\right| = 1$ as $\left|S^{\mathfrak{I}}\right| = c_S^{\mathfrak{I}}$ (for proof of $\left|S^{\mathfrak{I}}\right| = c_S^{\mathfrak{I}}$, see reasoning later in this proof for $|S| \approx c_S$ – the same reasoning works for all nodes $S \in V(\mathcal{G})$)

From, Rule 22 (SINGLETON), we know $x^{\mathfrak{S}} \in S^{\mathfrak{S}}$. By Proposition 2.10, $x^{\mathfrak{S}} \in S^{\mathfrak{S}}$. As

$$S^{\mathfrak{I}} = S^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(S)} B_t$$

and $\left|S^{\mathfrak{I}}\right| = 1$, we conclude that $S^{\mathfrak{I}} = \left\{x^{\mathfrak{S}}\right\} = \left\{x^{\mathfrak{I}}\right\}$.

4. $S \approx T \sqcup U$. We need to show $S^{\mathfrak{I}} = T^{\mathfrak{I}} \cup U^{\mathfrak{I}}$.

Let $S \notin V(\mathcal{G})$, $T \notin V(\mathcal{G})$, and $U \notin V(\mathcal{G})$. Then,

$$\begin{aligned} S^{\mathfrak{I}} &= S^{\mathfrak{S}} && (S \notin V(\mathcal{G})) \\ &= T^{\mathfrak{S}} \cup U^{\mathfrak{S}} && \text{(Proposition 2.10)} \\ &= T^{\mathfrak{I}} \cup U^{\mathfrak{I}} && (T \notin V(\mathcal{G}), U \notin V(\mathcal{G})) \end{aligned}$$

Otherwise, let $S \in V(\mathcal{G})$, or $T \in V(\mathcal{G})$, or $U \in V(\mathcal{G})$. Then, from Rules 31, 32, 33 and definition of $\mathrm{add}$, we know $S$, $T$, *and* $U$ in $V(\mathcal{G})$. Then,

$$\begin{aligned} S^{\mathfrak{I}} &= \bigcup_{t \in \mathcal{L}(S)} (t^{\mathfrak{S}} \cup B_t) && (S \in V(\mathcal{G})) \\ &= \bigcup_{t \in \mathcal{L}(T \sqcup U)} (t^{\mathfrak{S}} \cup B_t) && \text{(Proposition 2.11)} \\ &= \left( \bigcup_{t \in \mathcal{L}(T)} (t^{\mathfrak{S}} \cup B_t) \right) \cup \left( \bigcup_{t \in \mathcal{L}(U)} (t^{\mathfrak{S}} \cup B_t) \right) && \text{(Proposition 2.11)} \\ &= T^{\mathfrak{I}} \cup U^{\mathfrak{I}} && (T \in V(\mathcal{G}), U \in V(\mathcal{G})) \end{aligned}$$

5. $S \approx T \sqcap U$. We need to show $S^{\mathfrak{I}} = T^{\mathfrak{I}} \cap U^{\mathfrak{I}}$.

Let $S \notin V(\mathcal{G})$, $T \notin V(\mathcal{G})$, and $U \notin V(\mathcal{G})$. Then,

$$\begin{aligned} S^{\mathfrak{I}} &= S^{\mathfrak{S}} && (S \notin V(\mathcal{G})) \\ &= T^{\mathfrak{S}} \cap U^{\mathfrak{S}} && \text{(Proposition 2.10)} \\ &= T^{\mathfrak{I}} \cap U^{\mathfrak{I}} && (T \notin V(\mathcal{G}), U \notin V(\mathcal{G})) \end{aligned}$$

Let $S \notin V(\mathcal{G})$ and $T \notin V(\mathcal{G})$, but $U \in V(\mathcal{G})$. Then,

$$\begin{aligned} T^{\mathfrak{I}} \cap U^{\mathfrak{I}} &= T^{\mathfrak{S}} \cap U^{\mathfrak{I}} && (T \notin V(\mathcal{G})) \\ &= T^{\mathfrak{S}} \cap \left( U^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(U)} B_t \right) && (U \in V(\mathcal{G})) \\ &= T^{\mathfrak{S}} \cap U^{\mathfrak{S}} && (T^{\mathfrak{S}} \cap B_t = \emptyset) \\ &= S^{\mathfrak{S}} && \text{(Proposition 2.10)} \\ &= S^{\mathfrak{I}} && (S \notin V(\mathcal{G})) \end{aligned}$$

If $S \notin V(\mathcal{G})$ and $U \notin V(\mathcal{G})$, but $T \in V(\mathcal{G})$; the reasoning is same as above.

Otherwise, either $S \in V(\mathcal{G})$ or both $T \in V(\mathcal{G})$ and $U \in V(\mathcal{G})$. Then, from Rules 31, 32, 34 and definition of $\mathrm{add}$, we know $S$, $T$, *and* $U$ in $V(\mathcal{G})$. Then,

$$T^{\mathfrak{I}} \cap U^{\mathfrak{I}} = \left( T^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(T)} B_t \right) \cap \left( U^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(U)} B_t \right) \qquad (T, U \text{ in } V(\mathcal{G}))$$

As each $B_t$ is disjoint from all other sets, the above expression simplifies to:

$$\begin{aligned}
&= \left(T^{\mathfrak{S}} \cap U^{\mathfrak{S}}\right) \cup \bigcup_{t \in \mathcal{L}(T) \cap \mathcal{L}(U)} B_t \\
&= S^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(S)} B_t && \text{(Propositions 2.10 and 2.11)} \\
&= S^{\mathfrak{I}} && (S \in V(\mathcal{G}))
\end{aligned}$$

6. $S \approx T \setminus U$. We need to show $S^{\mathfrak{I}} = T^{\mathfrak{I}} \setminus U^{\mathfrak{I}}$.

   Let $S \notin V(\mathcal{G})$, $T \notin V(\mathcal{G})$, and $U \notin V(\mathcal{G})$. Then,

$$\begin{aligned}
S^{\mathfrak{I}} &= S^{\mathfrak{S}} && (S \notin V(\mathcal{G})) \\
&= T^{\mathfrak{S}} \setminus U^{\mathfrak{S}} && \text{(Proposition 2.10)} \\
&= T^{\mathfrak{I}} \setminus U^{\mathfrak{I}} && (T \notin V(\mathcal{G}), U \notin V(\mathcal{G}))
\end{aligned}$$

   Let $S \notin V(\mathcal{G})$ and $T \notin V(\mathcal{G})$, but $U \in V(\mathcal{G})$. Then,

$$\begin{aligned}
T^{\mathfrak{I}} \setminus U^{\mathfrak{I}} &= T^{\mathfrak{S}} \setminus U^{\mathfrak{I}} && (T \notin V(\mathcal{G})) \\
&= T^{\mathfrak{S}} \setminus \left(U^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(U)} B_t\right) && (U \in V(\mathcal{G})) \\
&= T^{\mathfrak{S}} \setminus U^{\mathfrak{S}} && (T^{\mathfrak{S}} \setminus B_t = T^{\mathfrak{S}}) \\
&= S^{\mathfrak{S}} && \text{(Proposition 2.10)} \\
&= S^{\mathfrak{I}} && (S \notin V(\mathcal{G}))
\end{aligned}$$

   Note that in contrast to intersection, if $S \notin V(\mathcal{G})$, $T \in V(\mathcal{G})$, and $U \notin V(\mathcal{G})$, the above analysis does not apply. We do need to introduce and reason about

the equality in the graph.

Let $S \in V(\mathcal{G})$ or $T \in V(\mathcal{G})$. From Rules 31, 32, 35 and definition of $\mathrm{add}$ we know $S, T$, *and* $U$ in $V(\mathcal{G})$. Then,

$$T^{\mathfrak{I}} \setminus U^{\mathfrak{I}} = \left(T^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(T)} B_t\right) \setminus \left(U^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(U)} B_t\right) \qquad (T, U \text{ in } V(\mathcal{G}))$$

As each $B_t$ is disjoint from all other sets, the above expression simplifies to:

$$\begin{aligned} &= \left(T^{\mathfrak{S}} \setminus U^{\mathfrak{S}}\right) \cup \bigcup_{t \in \mathcal{L}(T) \setminus \mathcal{L}(U)} B_t \\ &= S^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(S)} B_t && \text{(Propositions 2.10 and 2.11)} \\ &= S^{\mathfrak{I}} && (S \notin V(\mathcal{G})) \end{aligned}$$

7. $x \sqsubseteq S, x \not\sqsubseteq S$.

   Note that irrespective of whether $S \in V(\mathcal{G})$ or $S \notin V(\mathcal{G})$, $S^{\mathfrak{S}} \subseteq S^{\mathfrak{I}}$. Thus, from Proposition 2.10, $x^{\mathfrak{I}} \in S^{\mathfrak{I}}$ if $x \sqsubseteq S$ is a constraint.

   It remains to show that if $x \not\sqsubseteq S$ is a constraint then $x^{\mathfrak{I}} \notin S^{\mathfrak{I}}$. If $S \notin V(\mathcal{G})$, then again $x^{\mathfrak{I}} \notin S^{\mathfrak{I}}$ follows from Proposition 2.10. If $S \in V(\mathcal{G})$, then observe that $S^{\mathfrak{I}}$ is $S^{\mathfrak{S}} \cup \bigcup_{t \in \mathcal{L}(U)} B_t$. We already know $x^{\mathfrak{I}} \notin S^{\mathfrak{S}}$. It remains to show that $x^{\mathfrak{I}} \notin \bigcup_{t \in \mathcal{L}(U)} B_t$. This follows from the definition of $B_t$.

8. $c_S \approx |S|$.

   From Proposition 2.11, we know that for $t, u$ in $\mathcal{L}(S)$:

$$\models_{\mathfrak{T}_S} t \sqcap u \approx \emptyset$$

and also,

$$\models_{\mathfrak{T}_S} \left( \bigwedge_{t \in E} t \approx \emptyset \right) \Rightarrow \left( S \approx \bigsqcup_{t \in \mathcal{L}(S)} t \right)$$

where $E = \{t \in V(\mathcal{G}) \mid t \approx \emptyset \in \mathcal{S}^*\}$.

In $\mathfrak{I}$, as for each $t \in E$, $t^{\mathfrak{I}} = \emptyset$, it follows that:

$$S^{\mathfrak{I}} = \bigcup_{t \in \mathcal{L}(S)} t^{\mathfrak{I}} \quad .$$

Also, for $t, u$ in $\mathcal{L}(S)$:

$$t^{\mathfrak{I}} \cap u^{\mathfrak{I}} = \emptyset \quad .$$

In other words, $S^{\mathfrak{I}}$ is a disjoint union of $t^{\mathfrak{I}}$ where $t \in \mathcal{L}(S)$. It follows that,

$$\left|S^{\mathfrak{I}}\right| = \sum_{t \in \mathcal{L}(S)} \left|t^{\mathfrak{I}}\right|$$

For a leaf node $t \in \mathcal{L}(S)$, from (2.8) we know that $\left|t^{\mathfrak{I}}\right| = \left|t^{\mathfrak{S}}\right| + |B_t| = c_t^{\mathfrak{I}}$. We may thus conclude,

$$\left|S^{\mathfrak{I}}\right| = \sum_{t \in \mathcal{L}(S)} c_t^{\mathfrak{I}}$$

From the constraint on cardinality for $S$ induced by the graph, i.e the constraint on $c_S$ in $\hat{\mathcal{G}}$, we know that $c_S^{\mathfrak{I}} = \sum_{t \in \mathcal{L}(S)} c_t^{\mathfrak{I}}$. The result follows:

$$\left|S^{\mathfrak{I}}\right| = c_S^{\mathfrak{I}}$$

□

### 2.3.2 Soundness

Soundness of the rules 1-24, 26-28 follows trivially from the semantics of set operators and definition of $\mathcal{S}^*$. Soundness of Rule 25 (SET DISEQUALITY) also follows easily with a case analysis. Likewise for the optional propagation rule Rule 29 (SINGLETON AND UNION).

Soundness of Rule 38 (MERGE EQUALITY 1) and Rule 39 (MERGE EQUALITY 2) follows from Proposition 2.11 (in particular the property that leaf terms are disjoint). Rules 30-35 and 40-37 do not modify the constraints, but we need them to establish properties of the graph. Soundness of induced graph constraints in Rule 41 (ARITHMETIC CONTRADICTION) follows from properties of the Proposition 2.11 (in particular properties 5 and 6). Soundness of Rule 42 (GUESS EMPTY SET) is trivial.

Soundness of Rule 44 (PROPAGATE MINSIZE) follows from semantics of cardinality. Soundness of Rule 43 (MEMBERS ARRANGEMENT) and Rule 45 (GUESS LOWER BOUND) is trivial.

### 2.3.3 Termination

For the purpose of the following proposition, let $\mathcal{R}$ denote all rules in our calculus except the optional rules 29 and 45.

**Proposition 2.13** (Termination)**.** *Let $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{A}_0$ be normalized set, element, and arithmetic respectively. Let* **D** *be a derivation with respect to rules $\mathcal{R}$ starting from the state $\langle \mathcal{S}_0, \mathcal{M}_0, \mathcal{A}_0, (\emptyset, \emptyset)\rangle$. Then,* **D** *is necessarily finite.*

*Proof.* We will define a well-founded relation $\succ$ over states. Next, we will show that application of any rule in $\mathcal{R}$ to a leaf of a derivation tree gives smaller states with respect to this relation. As the relation is well-founded, it will follow that the derivation

cannot be infinite.

In order to define $\succ$, we define $f_i$ for $i \in \{1, 2, \ldots, 9\}$, each of which maps a state $\sigma = \langle \mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{G} \rangle$ to a natural number (non-negative integer). We denote the set of natural numbers by $\mathbb{N}$.

- $f_1(\sigma)$: number of equalities $t_1 \approx t_2$ in $\mathcal{S}$ such that either $t_1 \notin V(\mathcal{G})$, $t_2 \notin V(\mathcal{G})$, or $\mathcal{L}(t_1) \neq \mathcal{L}(t_2)$.

- $f_2(\sigma)$: size of $(\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S}) \cup \{\emptyset\}) \setminus V(\mathcal{G})$.

- $f_3(\sigma)$: size of $\{t \in \mathrm{Leaves}\,(\mathcal{G}) \mid t \approx \emptyset \notin \mathcal{S}^*, t \not\approx \emptyset \notin \mathcal{S}^*\}$.

- $f_4(\sigma)$: number of disequalities $t_1 \not\approx t_2$ in $\mathcal{S}$ such that the premise of Rule 25 (SET DISEQUALITY) holds.

- $f_5(\sigma)$: size of $\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S}) \cup \{\emptyset\} \cup V(\mathcal{G})$.

- $f_6(\sigma)$: size of $\mathrm{Terms}_{\mathsf{Element}}(\mathcal{S} \cup \mathcal{M})$.

- $f_7(\sigma)$: size of $\mathcal{M}^*$ subtracted from $2 \cdot (f_6(\sigma))^2$. As all constraints in $\mathcal{M}^*$ are either $x \approx y$ or $x \not\approx y$ with $x$ and $y$ in $\mathrm{Terms}_{\mathsf{Element}}(\mathcal{S} \cup \mathcal{M})$, the size of $\mathcal{M}^*$ can be at most $2 \cdot (f_6(\sigma))^2$. Thus, $f_7(\cdot)$ is well-defined as a map into $\mathbb{N}$.

- $f_8(\sigma)$: size of $\mathcal{S}^*$ subtracted from $2 \cdot (f_5(\sigma))^2 + 2 \cdot f_5(\sigma) \cdot f_6(\sigma)$. There are at most $2 \cdot (f_5(\sigma))^2$ constraints of the form $s \approx t$ or $s \not\approx t$ in $\mathcal{S}^*$ as $s$ and $t$ are in $\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S}) \cup \{\emptyset\} \cup V(\mathcal{G})$. There are at most $2 \cdot f_5(\sigma) \cdot f_6(\sigma)$ constraints of the form $x \sqsubseteq s$ or $x \not\sqsubseteq s$ in $\mathcal{S}^*$ as $x$ and $s$ are in $\mathrm{Terms}_{\mathsf{Element}}(\mathcal{S} \cup \mathcal{M})$ and $\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S}) \cup \{\emptyset\} \cup V(\mathcal{G})$ respectively. Thus, $f_8(\cdot)$ is well-defined as a map into $\mathbb{N}$.

- $f_9(\sigma)$: size of $(\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S}) \cup \{\emptyset\} \cup V(\mathcal{G})) \setminus \{t \in \mathrm{Leaves}\,(\mathcal{G}) \mid \mathcal{A} \not\Rightarrow c_t \geq |t_{\mathcal{S}}|\}$.

Then, we define the order $\succ$ over states as follows:

- $\sigma \succ \sigma'$ if $\sigma \neq \mathsf{unsat}$ and $\sigma' = \mathsf{unsat}$.
- $\sigma \succ \sigma'$ if $\sigma \neq \mathsf{unsat}$, $\sigma' \neq \mathsf{unsat}$, and

$$(f_1(\sigma), \ldots, f_9(\sigma)) >^9_{\mathsf{lex}} (f_1(\sigma'), \ldots, f_9(\sigma'))$$

  where $(\mathbb{N}^9, >^9_{\mathsf{lex}})$ is the 9-fold lexicographic product of ordering over natural numbers $(\mathbb{N}, >)$.

- $\sigma \not\succ \sigma'$ otherwise.

The well-foundedness of $\succ$ over states follows from the well-foundedness of $(\mathbb{N}^9, >^9_{\mathsf{lex}})$ [3, Section 2.4].

Let $r \in \mathcal{R}$ be a rule applicable at state $\sigma$, and let $\sigma'$ be the state after the application of the rule (if they are multiple conclusions denote the state on first branch as $\sigma'_1$, second branch as $\sigma'_2$ etc.). We note below for each rule $r \in \mathcal{R}$ the relation between $f_1(\sigma)$, ..., $f_9(\sigma)$ and $f_1(\sigma')$, ..., $f_9(\sigma')$ which establishes that $\sigma \succ \sigma'$.

- First, we consider Rules 1-7 for intersection, Rules 8-14 for union, Rules 15-21 for set difference, and Rule 22 for singleton. None of these rules introduce equalities of Set terms, nor do they affect the graph $\mathcal{G}$; thus $f_1(\sigma) \geq f_1(\sigma')$. The only terms introduced to $\mathcal{S}$ are from $V(\mathcal{G})$, thus $f_2(\sigma) = f_2(\sigma')$. None of these rules update $\mathcal{G}$ or introduce equalities or disequalities of Set terms, thus $f_3(\sigma) = f_3(\sigma')$. None of these rules introduce disequalities between Set terms, thus $f_4(\sigma) \geq f_4(\sigma')$. None of these rules introduce Set terms not already in $\mathcal{S}$ or $V(\mathcal{G})$, thus $f_5(\sigma) = f_5(\sigma')$. None of the rules introduce Element variables

not already in $\mathcal{S}$ or $\mathcal{M}$, thus $f_6(\sigma) = f_6(\sigma')$. None of these rules update $\mathcal{M}$, thus $f_7(\sigma) = f_7(\sigma')$.

Each of these rules updates $\mathcal{S}$. Recall that for a rule to be applicable at $\sigma$, the resulting state must be different from $\sigma$. From the definition of $\lhd$, we can conclude that size of $\mathcal{S}^*$ has increased. As $f_5(\sigma) = f_5(\sigma')$ and $f_6(\sigma) = f_6(\sigma')$, it follows that $f_8(\sigma) > f_8(\sigma')$.

- Next, we consider Rules 23, 24 and 43. None of these rules introduce equalities of Set terms, thus $f_1(\sigma) \geq f_1(\sigma')$. None of these rules introduce Set terms to $\mathcal{S}$ or $V(\mathcal{G})$, thus $f_2(\sigma) = f_2(\sigma')$. None of these rules update $\mathcal{G}$ or introduce equalities or disequality of Set terms, thus $f_3(\sigma) = f_3(\sigma')$. None of these rules introduce disequalities of Set terms, thus $f_4(\sigma) \geq f_4(\sigma')$. None of these rules introduce Set terms to $\mathcal{S}$ or $V(\mathcal{G})$, thus $f_5(\sigma) = f_5(\sigma')$. None of the rules introduce Element variables not already in $\mathcal{S}$ or $\mathcal{M}$, thus $f_6(\sigma) = f_6(\sigma')$.

  Each of these rules updates $\mathcal{M}$. From the definition of $\lhd$, we can conclude that size of $\mathcal{M}^*$ has increased. As $f_6(\sigma) = f_6(\sigma')$, we can conclude that $f_7(\sigma) > f_7(\sigma')$.

- Next, we consider Rule 25. The rule doesn't introduce any equality of Set terms, thus $f_1(\sigma) \geq f_1(\sigma'_i)$ for $i \in \{1, 2\}$. The rule doesn't introduce Set terms to $\mathcal{S}$ or $V(\mathcal{G})$, thus $f_2(\sigma) = f_2(\sigma'_i)$ for $i \in \{1, 2\}$. The rule doesn't update $\mathcal{G}$, thus $f_3(\sigma) \geq f_3(\sigma'_i)$ for $i \in \{1, 2\}$. The premise of the rule doesn't hold after application of the rule on either of the branches. It follows that $f_4(\sigma) > f_4(\sigma'_i)$ for $i \in \{1, 2\}$.

- Next, we consider Rules 30-37. None of these rules introduce equalities of Set terms, thus $f_1(\sigma) \geq f_1(\sigma')$.

Each of the rules adds at least one new node to $\mathcal{G}$ which is in $\mathrm{Terms}_{\mathsf{Set}}(\mathcal{S}) \cup \{\emptyset\}$. At the same time, $\mathcal{S}$ is unchanged. It follows that $f_2(\sigma) > f_2(\sigma')$.

- Rules 38 and 39. Though these rules add equalities of the form $u \approx \emptyset$ to $\mathcal{S}$, the equalities are such that $u \in V(\mathcal{G})$, $\emptyset \in V(\mathcal{G})$ and $\mathcal{L}(u) = \emptyset = \mathcal{L}(\emptyset)$. It follows that $f_1(\sigma) \geq f_1(\sigma')$.

  Now, observe that for Rule 38 or Rule 39 to be applicable, there must exist $s \approx t \in \mathcal{S}$ such that $\mathcal{L}(s) \neq \mathcal{L}(t)$. After the application of the rule, $\mathcal{L}(s) = \mathcal{L}(t)$. This shows that $f_1(\sigma) > f_1(\sigma')$.

- Rule 40. For the rule to be applicable, there must exist $s \approx t \in \mathcal{S}$ such that $\mathcal{L}(s) \neq \mathcal{L}(t)$. After the application of the rule, $\mathcal{L}(s) = \mathcal{L}(t)$. Thus, necessarily $f_1(\sigma) > f_1(\sigma')$.

- Rule 42. Note that though this rule may add an equality of the form $t \approx \emptyset$ on the first branch, using the same reasoning as for Rules 38 and 39 above, we can conclude that $f_1(\sigma) \geq f_1(\sigma'_1)$. On the second branch, as no disequality is added, we get that $f_1(\sigma) \geq f_1(\sigma'_2)$. Only terms introduced to $\mathcal{S}$ are from $V(\mathcal{G})$, thus $f_2(\sigma) = f_2(\sigma'_1)$ for $i \in \{1, 2\}$.

  In order to apply the rule, we pick a $t \in \mathrm{Leaves}\,(G)$ such that $t \approx \emptyset \notin \mathcal{S}^*$ and $t \not\approx \emptyset \notin \mathcal{S}^*$. On the first branch, $t \approx \emptyset \in \mathcal{S}^*$, thus $f_3(\sigma) > f_3(\sigma'_1)$. On the second branch, $t \not\approx \emptyset \in \mathcal{S}^*$, thus $f_3(\sigma) > f_3(\sigma'_2)$.

- Rule 44. The rule doesn't update $\mathcal{S}$, $\mathcal{M}$, or $\mathcal{G}$, thus $f_1(\sigma) = f_1(\sigma')$, $f_2(\sigma) = f_2(\sigma')$, $f_3(\sigma) = f_3(\sigma')$, $f_4(\sigma) = f_4(\sigma')$, $f_5(\sigma) = f_5(\sigma')$, $f_6(\sigma) = f_6(\sigma')$, $f_7(\sigma) = f_7(\sigma')$, and $f_8(\sigma) = f_8(\sigma')$. But, $f_9(\sigma) > f_9(\sigma')$.

- Rules 26, 27, 28, and 41. For each of these rules to be applicable, $\sigma \neq \mathsf{unsat}$.

On the other hand, $\sigma' = \mathsf{unsat}$ after the application of the rule. By definition, $\sigma \succ \sigma'$.

As $\succ$ order over states is well-founded, and application of any rule gives a smaller state with respect to this order, the derivation $\mathbf{D}$ must be necessarily finite.

*Remark.* It is easy to extend the termination proof to include the optional rules, like Rules 29 and 45. It would involve tracking sizes of additional objects. For instance, for Rule 29, tracking the number of singleton set terms for which the rule has not yet been applied. Since no rule introduces new singleton set terms, this can only go down. For Rule 45, a strategy similar to one adopted for Rule 42 in our proof would suffice.

□

In Chapter 5, we discuss how we can use the calculus to obtain an efficient, incremental decision procedure for the theory in an SMT solver.

## 2.4 Related work

In this section we compare the calculus with work which is most closely related to the fragment considered in this chapter.

In [58], the decidability of this fragment was established. The procedure involves making an up-front guess that is exponential in the number of set variables in the input, making it non-incremental and highly impractical for reasoning in an SMT solver. That said, the focus of [58] is on establishing decidability, and not on providing an efficient procedure.

Another logical fragment that is closely related is that of Boolean Alegbra and Presburger Arithmetic (BAPA), for which several algorithms have been proposed

[39, 40, 56]. Though the fragment doesn't have membership predicate and singleton operator in the language, [56, Section 4] discusses how one can generalize the algorithm for such reasoning. Intutively, the idea is to simulate singleton sets by imposing the constraint $|X| = 1$ and then writing a membership constraint, say $x \sqsubseteq S$, by introducing a singleton set for it, say $X$, and using the subset operation, e.g. $X \sqsubseteq S$.

To illustrate a potential down-side of this approach, consider a simple example: $x \sqsubseteq S_1 \sqcup (S_2 \sqcup (\ldots \sqcup (S_{99} \sqcup S_{100})))$, $x \not\sqsubseteq S_1$, $x \not\sqsubseteq S_2$, ..., $x \not\sqsubseteq S_{100}$. It is easy to see the set of constraints is unsatisfiable. In our calculus, a straightforward application of the (propagation) Rule 11 will derive unsat. On the other hand, in a reduction to BAPA, the membership reasoning if reduced to reasoning about cardinality of different sets, such reasoning becomes inefficient. For instance, in [56], the algorithm will reduce the reasoning to arithmetic constraints involving variables for $2^{101}$ Venn regions derived from $S_1$, $S_2$, ..., $S_{100}$ and the singleton set introduced for $x$.

The broader point is that reasoning about membership predicates as is done by the rules in Section 2.2.1 indirectly via cardinalities of Venn regions is inefficient. As we show in our calculus, it is possible to avoid this. We reason about membership directly, and minimize the number of regions whose cardinality we reason about. The latter is done by being careful about terms we introduce to the graph used for cardinality reasoning, and breaking down regions incrementally.

## 2.5 Conclusion

In this chapter, we presented a calculus for the theory of finite sets with cardinality constraints and proved its correctness. In Chapter 5, we discuss how we can use the calculus to obtain an efficient, incremental decision procedure for the theory in an

SMT solver. We also share our experience with implementation and experimental results.

# Chapter 3

# Local theory extensions

One of the appeal of SMT solvers, as discussed earlier, is that they implement decision procedures for efficiently reasoning about formulas in theories that model the semantics of prevalent data types and software constructs. Examples include integers, bitvectors, arrays, and with work in previous chapter expanding it to sets. But, some verification tasks involve reasoning about universally quantified formulas, which goes beyond the capabilities of the solvers' core decision procedures. Typical examples include verification of programs with complex data structures and concurrency, yielding formulas that quantify over unbounded sets of memory locations or thread identifiers.

From a logical perspective, these quantified formulas can be thought of as axioms of application-specific theories. In practice, such theories often remain within decidable fragments of first-order logic [1, 11, 13, 33]. However, their narrow scope (which is typically restricted to a specific program) does not justify the implementation of a dedicated decision procedure inside the SMT solver. Instead, many solvers allow theory axioms to be specified directly in the input constraints. The solver then provides a quantifier module that is designed to heuristically instantiate these axioms. These

heuristics are in general incomplete and the user is given little control over the instance generation. Thus, even if there exists a finite instantiation strategy that yields a decision procedure for a specific set of axioms, the communication of strategies and tactics to SMT solvers is a challenge [17]. Further, the user cannot communicate the completeness of such a strategy. In this situation, the user is left with two alternatives: either she gives up on completeness, which may lead to usability issues in the verification tool, or she implements her own instantiation engine as a preprocessor to the SMT solver, leading to duplication of effort and reduced solver performance.

The contributions of this chapter are two-fold. First, we provide a better understanding of how complete decision procedures for application-specific theories can be realized with the quantifier modules that are implemented in SMT solvers. Second, we explore several extensions of the capabilities of these modules to better serve the needs of verification tool developers. The focus of our exploration is on *local theory extensions* [31, 54]. A theory extension extends a given base theory with additional symbols and axioms. Local theory extensions are a class of such extensions that can be decided using finite quantifier instantiation of the extension axioms. This class is attractive because it is characterized by proof and model-theoretic properties that abstract from the intricacies of specific quantifier instantiation techniques [22, 30, 54]. Also, many well-known theories that are important in verification but not commonly supported by SMT solvers are in fact local theory extensions, even if they have not been presented as such in the literature. Examples include the array property fragment [12], the theory of reachability in linked lists [41, 49], and the theories of finite sets [59] and multisets [57].

We present a general decision procedure for local theory extensions that relies on E-matching, one of the core components of the quantifier modules in SMT solvers,

which will be discussed in this chapter. We have implemented our decision procedure using the SMT solvers CVC4 [8] and Z3 [16] and applied it to a large set of SMT benchmarks coming from the deductive software verification tool GRASShopper [46, 48]. We will discuss the experimental results in detail in Chapter 5. These benchmarks use a hierarchical combination of local theory extensions to encode verification conditions that express correctness properties of programs manipulating complex heap-allocated data structures. Guided by our experiments, we developed generic optimizations in CVC4 that improve the performance of our base-line decision procedure. Some of these optimizations required us to implement extensions in the solver's quantifier module. We believe that our results are of interest to both the users of SMT solvers as well as their developers. For users we provide simple ways of realizing complete decision procedures for application-specific theories with today's SMT solvers. For developers we provide interesting insights that can help them further improve the completeness and performance of today's quantifier instantiation modules.

## 3.1 Background

### 3.1.1 Example

Sofronie-Stokkermans [54] introduced local theory extensions as a generalization of locality in equational theories [22, 25]. We start our discussion with a simple example that illustrates the basic idea behind local theory extensions. Consider the following set of ground literals

$$G = \{a + b = 1, f(a) + f(b) = 0\}.$$

We interpret $G$ in the theory of linear integer arithmetic and a monotonically increasing function $f : \mathbb{Z} \to \mathbb{Z}$. One satisfying assignment for $G$ is:

$$a = 0, b = 1, f(x) = \{-1 \text{ if } x \leq 0, 1 \text{ if } x > 0\}. \tag{3.1}$$

We now explain how we can use an SMT solver to conclude that $G$ is indeed satisfiable in the above theory.

SMT solvers commonly provide inbuilt decision procedures for common theories such as the theory of linear integer arithmetic (`LIA`) and the theory of equality over uninterpreted functions (`UF`). However, they do not natively support the theory of monotone functions. The standard way to enforce $f$ to be monotonic is to axiomatize this property,

$$K = \forall x, y.\, x \leq y \implies f(x) \leq f(y), \tag{3.2}$$

and then let the SMT solver check if $G \cup \{K\}$ is satisfiable via a reduction to its natively supported theories. In our example, the reduction target is the combination of `LIA` and `UF`, which we refer to as the *base theory*, denoted by $\mathcal{T}_0$. We refer to the axiom $K$ as a *theory extension* of the base theory and to the function symbol $f$ as an *extension symbol*.

Most SMT solvers divide the work of deciding ground formulas $G$ in a base theory $\mathcal{T}_0$ and axioms $\mathcal{K}$ of theory extensions between different modules. A quantifier module looks for substitutions to the variables within an axiom $K$, $x$ and $y$, to some ground terms, $t_1$ and $t_2$. We denote such a substitution as $\sigma = \{x \mapsto t_1, y \mapsto t_2\}$ and the instance of an axiom $K$ with respect to this substitution as $K\sigma$. The quantifier module iteratively adds the generated ground instances $K\sigma$ as lemmas to $G$ until the base theory solver derives a contradiction. However, if $G$ is satisfiable, as in our case,

then the quantifier module does not know when to stop generating instances of $K$, and the solver may diverge, effectively enumerating an infinite model of $G$.

For a local theory extension, we can syntactically restrict the instances $K\sigma$ that need to be considered before concluding that $G$ is satisfiable to a finite set of candidates. More precisely, a theory extension is called *local* if in order to decide satisfiability of $G \cup \{K\}$, it is sufficient to consider only those instances $K\sigma$ in which all ground terms already occur in $G$ and $K$. The monotonicity axiom $K$ is a local theory extension of $\mathcal{T}_0$. The local instances of $K$ and $G$ are:

$$
\begin{aligned}
K\sigma_1 = a \leq b \implies f(a) \leq f(b) \text{ where } \sigma_1 = \{x \mapsto a, y \mapsto b\},\\
K\sigma_2 = b \leq a \implies f(b) \leq f(a) \text{ where } \sigma_2 = \{x \mapsto b, y \mapsto a\},\\
K\sigma_3 = a \leq a \implies f(a) \leq f(a) \text{ where } \sigma_3 = \{x \mapsto a, y \mapsto a\}, \text{ and}\\
K\sigma_4 = b \leq b \implies f(b) \leq f(b) \text{ where } \sigma_4 = \{x \mapsto b, y \mapsto b\}.
\end{aligned}
$$

Note that we do not need to instantiate $x$ and $y$ with other ground terms in $G$, such as $0$ and $1$. Adding the above instances to $G$ yields

$$G' = G \cup \{K\sigma_1, K\sigma_2, K\sigma_3, K\sigma_4\}.$$

which is satisfiable in the base theory. Since $K$ is a local theory extension, we can immediately conclude that $G \cup \{K\}$ is also satisfiable.

### 3.1.2 Semantic characterizations

There are two useful characterizations of local theory extensions that can help users of SMT solvers in designing axiomatization that are local. The first one is model-

theoretic [22, 54]. Consider again the set of ground clauses $G'$. When checking satisfiability of $G'$ in the base theory, the SMT solver may produce the following model:

$$a = 0, b = 1, f(x) = \{-1 \text{ if } x = 0, 1 \text{ if } x = 1, \text{-1 otherwise}\}. \tag{3.3}$$

This is not a model of the original $G \cup \{K\}$. However, if we restrict the interpretation of the extension symbol $f$ in this model to the ground terms in $G \cup \{K\}$, we obtain the *partial model*

$$a = 0, b = 1, f(x) = \{-1 \text{ if } x = 0, 1 \text{ if } x = 1, \text{undefined otherwise}\}. \tag{3.4}$$

This partial model can now be embedded into the model (3.1) of the theory extension. If such embeddings of partial models of $G'$ to total models of $G \cup \{K\}$ always exist for all sets of ground literals $G$, then $K$ is a local theory extension of $\mathcal{T}_0$. The second characterization of local theory extensions is proof-theoretic and states that a set of axioms is a local theory extension if it is saturated under (ordered) resolution [10]. This characterization can be used to automatically compute local theory extensions from non-local ones [30].

Note that the locality property depends both on the base theory as well as the specific axiomatization of the theory extension. For example, the following axiomatization of a monotone function $f$ over the integers, which is logically equivalent to equation (3.2) in $\mathcal{T}_0$, is not local:

$$K = \forall x.\, f(x) \leq f(x+1) \ .$$

Similarly, if we replace all inequalities in equation (3.2) by strict inequalities, then the

extension is no longer local for the base theory $\mathcal{T}_0$. However, if we replace $\mathcal{T}_0$ by a theory in which $\leq$ is a dense order (such as in linear real arithmetic), then the strict version of the monotonicity axiom is again a local theory extension.

In the remaining part of the chapter, we show how we can use the existing technology implemented in quantifier modules of SMT solvers to decide local theory extensions. In particular, we show how E-matching can be used to further reduce the number of axiom instances that need to be considered before we can conclude that a given set of ground literals $G$ is satisfiable.

## 3.2 Formal definition

In order to keep each chapter self-contained, we recall the basic notions of first-order logic, and additional terminology needed for this chapter.

### Sorted first-order logic

We present our problem in sorted first-order logic with equality. A *signature* $\Sigma$ is a tuple $(\mathsf{Sorts}, \Omega, \Pi)$, where $\mathsf{Sorts}$ is a countable set of sorts and $\Omega$ and $\Pi$ are countable sets of function and predicate symbols, respectively. Each function symbol $f \in \Omega$ has an associated arity $n \geq 0$ and associated sort $s_1 \times \cdots \times s_n \to s_0$ with $s_i \in \mathsf{Sorts}$ for all $i \leq n$. Function symbols of arity 0 are called *constant symbols*. Similarly, predicate symbols $P \in \Pi$ have an arity $n \geq 0$ and sort $s_1 \times \cdots \times s_n$. We assume dedicated equality symbols $\approx_s \in \Pi$ with the sort $s \times s$ for all sorts $s \in \mathsf{Sorts}$, though we typically drop the explicit subscript. Terms are built from the function symbols in $\Omega$ and (sorted) variables taken from a countably infinite set $X$ that is disjoint from $\Omega$. We denote by $t : s$ that term $t$ has sort $s$.

A $\Sigma$-atom $A$ is of the form $P(t_1, \ldots, t_n)$ where $P \in \Pi$ is a predicate symbol of sort $s_1 \times \cdots \times s_n$ and the $t_i$ are terms with $t_i : s_i$. A $\Sigma$-*formula* $F$ is either a $\Sigma$-atom $A$, $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$, or $\forall x : s.F_1$ where $F_1$ and $F_2$ are $\Sigma$-formulas. A $\Sigma$-*literal* $L$ is either $A$ or $\neg A$ for a $\Sigma$-atom $A$. A $\Sigma$-*clause* $C$ is a disjunction of $\Sigma$-literals. A $\Sigma$-term, atom, or formula is said to be *ground*, if no variable appears in it. For a set of formulas $\mathcal{K}$, we denote by $\mathsf{st}(\mathcal{K})$ the set of all ground subterms that appear in $\mathcal{K}$.

A $\Sigma$-sentence is a $\Sigma$-formula with no free variables where the free variables of a formula are defined in the standard fashion. We typically omit $\Sigma$ if it is clear from the context.

### Structures

Given a signature $\Sigma = (\mathsf{Sorts}, \Omega, \Pi)$, a $\Sigma$-*structure* $M$ is a function that maps each sort $s \in \mathsf{Sorts}$ to a non-empty set $M(s)$, each function symbol $f \in \Omega$ of sort $s_1 \times \cdots \times s_n \to s_0$ to a function $M(f) : M(s_1) \times \cdots \times M(s_n) \to M(s_0)$, and each predicate symbol $P \in \Pi$ of sort $s_1 \times \cdots \times s_n$ to a relation $M(s_1) \times \cdots \times M(s_n)$. We assume that all structures $M$ interpret each symbol $\approx_s$ by the equality relation on $M(s)$. For a $\Sigma$-structure $M$ where $\Sigma$ extends a signature $\Sigma_0$ with additional sorts and function symbols, we write $M|_{\Sigma_0}$ for the $\Sigma_0$-structure obtained by restricting $M$ to $\Sigma_0$.

Given a structure $M$ and a *variable assignment* $\nu : X \to M$, the evaluation $t^{M,\nu}$ of a term $t$ in $M, \nu$ is defined as usual. For a structure $M$ and an atom $A$ of the form $P(t_1, \ldots, t_n)$, $(M, \nu)$ satisfies $A$ iff $(t_1^{M,\nu}, \ldots, t_n^{M,\nu}) \in M(P)$. This is written as $(M, \nu) \models A$. From this satisfaction relation of atoms and $\Sigma$-structures, we can derive the standard notions of the satisfiability of a formula, satisfying a set of formulas $(M, \nu) \models \{F_i\}$, validity $\models F$, and entailment $F_1 \models F_2$. If a $\Sigma$-structure $M$ satisfies a $\Sigma$-sentence $F$, we call $M$ a model of $F$.

### 3.2.1 Theory extensions

A *theory* $\mathcal{T}$ over signature $\Sigma$ is a set of $\Sigma$-structures. We call a $\Sigma$-sentence $K$ an *axiom* if it is the universal closure of a $\Sigma$-clause, and we denote a set of $\Sigma$-axioms as $\mathcal{K}$. We consider theories $\mathcal{T}$ defined as a class of $\Sigma$-structures that are models of a given set of $\Sigma$-sentences $\mathcal{K}$.

**Definition 3.1** (Theory extension)**.** *Let* $\Sigma_0 = (\mathsf{Sorts}_0, \Omega_0, \Pi)$ *be a signature and assume that the signature* $\Sigma_1 = (\mathsf{Sorts}_0 \cup \mathsf{Sorts}_e, \Omega_0 \cup \Omega_e, \Pi)$ *extends* $\Sigma_0$ *by new sorts* $\mathsf{Sorts}_e$ *and function symbols* $\Omega_e$*. Given a* $\Sigma_0$*-theory* $\mathcal{T}_0$ *and* $\Sigma_1$*-axioms* $\mathcal{K}_e$*, we call* $(\mathcal{T}_0, \mathcal{K}_e, \mathcal{T}_1)$ *the* theory extension *of* $\mathcal{T}_0$ *with* $\mathcal{K}_e$*, where* $\mathcal{T}_1$ *is the set of all* $\Sigma_1$*-structures* $M$ *that are models of* $\mathcal{K}_e$ *and whose reducts* $M|_{\Sigma_0}$ *are in* $\mathcal{T}_0$*.*

We call the elements of $\Omega_e$ *extension symbols* and terms starting with extension symbols *extension terms*. We often identify the theory extension with the theory $\mathcal{T}_1$.

### 3.2.2 Local theories and satisfiability problem

We formally define the problem of satisfiability modulo theory and the notion of local theory extensions in this section.

Let $\mathcal{T}$ be a theory over signature $\Sigma$. Given a $\Sigma$-formula $\phi$, we say $\phi$ is satisfiable modulo $\mathcal{T}$ if there exists a structure $M$ in $\mathcal{T}$ and an assignment $\nu$ of the variables in $\phi$ such that $(M, \nu) \models \phi$. We define the ground satisfiability modulo theory problem as the corresponding decision problem for quantifier-free formulas.

**Problem 3.2** (Ground satisfiability problem for $\Sigma$-theory $\mathcal{T}$)**.**

**input:** *A quantifier-free* $\Sigma$*-formula* $\phi$*.*

**output:** *sat if* $\phi$ *is satisfiable modulo* $\mathcal{T}$*, unsat otherwise.*

We say the satisfiability problem for $\mathcal{T}$ is *decidable* if there exists a procedure for the above problem which always terminates with sat or unsat. We write entailment modulo a theory as $\phi \models_{\mathcal{T}} \psi$.

We say an axiom of a theory extension is *linear* if all the variables occur under at most one extension term. We say it is *flat* if there there is no nesting of terms containing variables. It is easy to linearize and flatten the axioms by using additional variables and equality. As an example, $\forall x.\phi$ with $f(x)$ and $f(g(x))$ as terms in $F$ may be written as

$$\forall xyz.x \approx y \wedge z \approx g(y) \implies F'$$

where $F'$ is obtained from $F$ by replacing $f(g(x))$ with $f(z)$. For the remainder of the chapter, we assume that all extension axioms $\mathcal{K}_e$ are flat and linear. For the simplicity of the presentation, we assume that if a variable appears below a function symbol then that symbol must be an extension symbol.

**Definition 3.3** (Local theory extensions)**.** *A theory extension* $(\mathcal{T}_0, \mathcal{K}_e, \mathcal{T}_1)$ *is* local *if for any set of ground* $\Sigma_1$*-literals* $G$*:* $G$ *is satisfiable modulo* $\mathcal{T}_1$ *if and only if* $G \cup \mathcal{K}_e[G]$ *is satisfiable modulo* $\mathcal{T}_0$ *extended with free function symbols. Here* $\mathcal{K}_e[G]$ *is the set of instances of* $\mathcal{K}_e$ *where the subterms of the instantiation are all subterms of* $G$ *or* $\mathcal{K}_e$ *(in other words, they do not introduce new terms).*

For simplicity, in the rest of this chapter, we work with theories $\mathcal{T}_0$ which have decision procedures for not just $\mathcal{T}_0$ but also $\mathcal{T}_0$ extended with free function symbols. Thus, we sometimes talk of satisfiability of a $\Sigma_1$-formula with respect a $\Sigma_0$-theory $\mathcal{T}_0$, to mean satisfiability in the $\mathcal{T}_0$ with the extension symbols in $\Sigma_1$ treated as free function symbols. In terms of SMT, we only talk of extensions of theories containing uninterpreted functions (UF).

A naive decision procedure for ground SMT of a local theory extension $\mathcal{T}_1$ is thus to generate all possible instances of the axioms $\mathcal{K}_e$ that do not introduce new ground terms, thereby reducing to the ground SMT problem of $\mathcal{T}_0$ extended with free functions.

**Hierarchical extensions.**

Note that local theory extensions can be stacked to form hierarchies $((\ldots( (\mathcal{T}_0, \mathcal{K}_1, \mathcal{T}_1), \mathcal{K}_2, \mathcal{T}_2), \ldots), \mathcal{K}_n, \mathcal{T}_n)$. Such a hierarchical arrangement of extension axioms is often useful to modularize locality proofs. In such cases, the condition that variables are only allowed to occur below extension symbols (of the current extension) can be relaxed to any extension symbol of the current level or below. The resulting theory extension can be decided by composing procedures for the individual extensions. Alternatively, one can use a monolithic decision procedure for the resulting theory $\mathcal{T}_n$, which can also be viewed as a single local theory extension $(\mathcal{T}_0, \mathcal{K}_1 \cup \cdots \cup \mathcal{K}_n, \mathcal{T}_n)$. In our experimental evaluation, which involved hierarchical extensions, we followed the latter approach.

## 3.3 Algorithm

In this section, we describe a decision procedure for a local theory extension, say $(\mathcal{T}_0, \mathcal{K}_e, \mathcal{T}_1)$, which can be easily implemented in most SMT solvers with quantifier instantiation support. We describe our procedure $\mathfrak{D}_{\mathcal{T}_1}$ as a theory module in a typical SMT solver architecture. For simplicity, we separate out the interaction between theory solver and core SMT solver. We describe the procedure abstractly as taking as input:

- the original formula $\phi$,
- a set of extension axioms $\mathcal{K}_e$,
- a set of instantiations of axioms that have already been made, $Z$, and
- a set of $\mathcal{T}_0$ satisfiable ground literals $G$ such that $G \models \phi \wedge (\bigwedge_{\psi \in Z} \psi)$, and
- a set equalities $E \subseteq G$ between terms.

It either returns

- sat, denoting that $G$ is $\mathcal{T}_1$ satisfiable; or
- a new set of instantiations of the axioms, $Z'$.

For completeness, we describe briefly the way we envisage the interaction mechanism of this module in a DPLL(T) SMT solver. Let the input problem be $\phi$. The SAT solver along with the theory solvers for $\mathcal{T}_0$ will find a subset of literals $G$ from $\phi \wedge (\bigwedge_{\psi \in Z} \psi)$ such that its conjunction is satisfiable modulo $\mathcal{T}_0$. If no such satisfying assignment exists, the SMT solver stops with unsat. One can think of $G$ as being simply the literals in $\phi$ on the SAT solver trail. $G$ will be sent to $\mathfrak{D}_{\mathcal{T}_1}$ along with information known about equalities between terms. The set $Z$ can be also thought of as internal state maintained by the $\mathcal{T}_1$-theory solver module, with new instances $Z'$ sent out as theory lemmas and $Z$ updated to $Z \cup Z'$ after each call to $\mathfrak{D}_{\mathcal{T}_1}$. If $\mathfrak{D}_{\mathcal{T}_1}$ returns sat, so does the SMT solver and stops. On the other hand, if it returns a new set of instances, the SMT solver continues the search to additionally satisfy these.

*E-matching.* In order to describe our procedure, we introduce the well-studied E-matching problem. Given a universally quantified $\Sigma$-sentence $K$, let $X(K)$ denote

the quantified variables. Define a $\Sigma$-substitution $\sigma$ of $K$ to be a mapping from variables $X(K)$ to $\Sigma$-terms of corresponding sort. Given a $\Sigma$-term $p$, let $p\sigma$ denote the term obtained by substituting variables in $p$ by the substitutions provided in $\sigma$. Two substitutions $\sigma_1$, $\sigma_2$ with the same domain $X$ are equivalent modulo a set of equalities $E$ if $\forall x \in X.\, E \models \sigma_1(x) \approx \sigma_2(x)$. We denote this as $\sigma_1 \sim_E \sigma_2$.

**Problem 3.4** (E-matching)**. input:** *A set of ground equalities $E$, a set of $\Sigma$-terms $G$, and patterns $P$.*

**output:** *The set of substitutions $\sigma$ over the variables in $p$, modulo $E$, such that for all $p \in P$ there exists a $t \in G$ with $E \models t \approx p\sigma$.*

E-matching is a well-studied problem, specifically in the context of SMT. An algorithm for E-matching that is efficient and backtrackable is described in [15]. We denote this procedure by $\mathfrak{E}$.

The procedure $\mathfrak{D}_{\mathcal{T}_1}(\phi, \mathcal{K}_e, Z, G, E)$ is given in Fig. 3.1. Intuitively, it adds all the new instances along the current search path that are required for local theory reasoning as given in Definition 3.3, but modulo equality. For each axiom $K$ in $\mathcal{K}_e$, the algorithm looks for function symbols containing variables. For example, if we think of the monotonicity axiom in Sect. 3.1.1, these would be the terms $f(x)$ and $f(y)$. These terms serve as patterns for the E-matching procedure. Next, with the help of the E-matching algorithm, all *new* instances are computed (to be more precise, all instances for the axiom $K$ in $Z$ which are equivalent modulo $\sim_E$ are skipped). If there are no new instances for any axiom in $\mathcal{K}_e$, and the set $G$ of literals implies $\phi$, we stop with sat. as effectively we have that $G \cup \mathcal{K}_e[G]$ is satisfiable modulo $\mathcal{T}_0$. Otherwise, we return this set.

We note that though the algorithm $\mathfrak{D}_{\mathcal{T}_1}$ may *look* inefficient because of the pres-

$\mathfrak{D}_{\mathcal{T}_1}(\phi, \mathcal{K}_e, Z, G, E)$
Local variable: $Z'$, initially an empty set.

1. For each $K \in \mathcal{K}_e$:

   (a) Define the set of patterns $P$ to be the function symbols in $K$ containing variables. We observe that because the axioms are linear and flat, these patterns are always of the form $f(x_1, \ldots, x_n)$ where $f$ is an extension symbol and the $x_i$ are quantified variables.

   (b) Run $\mathfrak{E}(E, G, P)$ obtaining substitutions $\mathcal{S}$. Without loss of generality, assume that $\sigma \in \mathcal{S}$ returned by the algorithm are such that $\mathsf{st}(K\sigma) \subseteq \mathsf{st}(G \cup \mathcal{K}_e)$. For the special case of the patterns in (a), for any $\sigma$ not respecting the condition there exists one in the equivalence class that respects the condition. Formally, $\forall\sigma.\exists\sigma'.\sigma' \sim_E \sigma \wedge \mathsf{st}(K\sigma') \subseteq \mathsf{st}(G \cup \mathcal{K}_e)$. We make this assumption only for simplicity of arguments later in the chapter. If one uses an E-matching procedure not respecting this constraint, our procedure will still be terminating and correct (albeit total number of instantiations suboptimal).

   (c) For each $\sigma \in \mathcal{S}$, if there exists no $K\sigma'$ in $Z$ such that $\sigma \sim_E \sigma'$, then add $K\sigma$ to $Z'$ as a new instantiation to be made.

2. If $Z'$ is empty, return sat, else return $Z'$.

Figure 3.1: Procedure $\mathfrak{D}_{\mathcal{T}_1}$

ence of nested loops, keeping track of which substitutions have already happened, and which substitutions are new. However, in actual implementations all of this is taken care of by the E-matching algorithm. There has been significant research on fast, incremental algorithms for E-matching in the context of SMT, and one advantage of our approach is to be able to leverage this work.

### 3.3.1 Correctness

The correctness argument relies on two aspects: one, that if the SMT solver says sat (resp. unsat) then $\phi$ is satisfiable (resp. unsatisfiable) modulo $\mathcal{T}_1$, and second, that it

terminates.

For the case where the output is unsat, the correctness follows from the fact that $Z$ only contains instances of $\mathcal{K}_e$. The sat case is more tricky, but the main idea is that the set of instances made by $\mathfrak{D}_{\mathcal{T}_1}(\phi, \mathcal{K}_e, Z, G, E)$ are logically equivalent to $\mathcal{K}_e[G]$. Thus, when the solver stops, $G \cup \mathcal{K}_e[G]$ is satisfiable modulo $\mathcal{T}_0$. As a consequence, $G$ is satisfiable modulo $\mathcal{T}_1$. Since $G \models \phi$, we have that $\phi$ is satisfiable modulo $\mathcal{T}_1$.

The termination relies on the fact that the instantiations returned by procedure $\mathfrak{D}_{\mathcal{T}_1}(\phi, \mathcal{K}_e, Z, G, E)$ is logically equivalent to an instantiation in $\mathcal{K}_e[\phi]$. Since, $\mathcal{K}_e[\phi]$ is finite, eventually $\mathfrak{D}$ will stop making new instantiations. Assuming that we have a terminating decision procedure for the ground SMT problem of $\mathcal{T}_0$, we get a terminating decision procedure for $\mathcal{T}_1$.

**Theorem 3.5.** *An SMT solver with theory module $\mathfrak{D}_{\mathcal{T}_1}$ is a decision procedure for the satisfiability problem modulo $\mathcal{T}_1$.*

### 3.3.2 Psi-local theories

We briefly explain how our approach can be extended to the more general notion of Psi-local theory extensions [31]. Sometimes, it is not sufficient to consider only local instances of extension axioms to decide satisfiability modulo a theory extension. For example, consider the following set of ground literals:

$$G = \{f(a) = f(b), a \neq b\}$$

Suppose we interpret $G$ in a theory of an injective function $f : S \to S$ with a partial inverse $g : S \to S$ for some set $S$. We can axiomatize this theory as a theory extension

of the theory of uninterpreted functions using the axiom

$$K = \forall x, y.\, f(x) = y \implies g(y) = x \ .$$

$G$ is unsatisfiable in the theory extension, but the local instances of $K$ with respect to the ground terms $\mathsf{st}(G) = \{a, b, f(a), f(b)\}$ are insufficient to yield a contradiction in the base theory. However, if we consider the local instances with respect to the larger set of ground terms

$$\Psi(\mathsf{st}(G)) = \{a, b, f(a), f(b), g(f(a)), g(f(b))\},$$

then we obtain, among others, the instances

$$f(a) = f(b) \implies g(f(b)) = a \quad \text{and} \quad f(b) = f(a) \implies g(f(a)) = b \ .$$

Together with $G$, these instances are unsatisfiable in the base theory.

The set $\Psi(\mathsf{st}(G))$ is computed as follows:

$$\Psi(\mathsf{st}(G)) = \mathsf{st}(G) \cup \{\, g(f(t)) \mid t \in \mathsf{st}(G) \,\}$$

It turns out that considering local instances with respect to $\Psi(\mathsf{st}(G))$ is sufficient to check satisfiability modulo the theory extension $K$ for arbitrary sets of ground clauses $G$. Moreover, $\Psi(\mathsf{st}(G))$ is always finite. Thus, we still obtain a decision procedure for the theory extension via finite instantiation of extension axioms. Psi-local theory extensions formalize this idea. In particular, if $\Psi$ satisfies certain properties including monotonicity and idempotence, one can again provide a model-theoretic characteri-

zation of completeness in terms of embeddings of partial models. We refer the reader to [31] for the technical details.

To use our algorithm for deciding satisfiability of a set of ground literals $G$ modulo a Psi-local theory extension $(\mathcal{T}_0, \mathcal{K}_e, \mathcal{T}_1)$, we simply need to add an additional preprocessing step in which we compute $\Psi(\mathsf{st}(G))$ and define $G' = G \cup \{\, \mathtt{instclosure}(t) \mid t \in \Psi(\mathsf{st}(G)) \,\}$ where `instclosure` is a fresh predicate symbol. Then calling our procedure for $\mathcal{T}_1$ with $G'$ decides satisfiability of $G$ modulo $\mathcal{T}_1$.

## 3.4 Conclusion

In this chapter, we presented a new algorithm for deciding local theory extensions, a class of theories that plays an important role in verification applications. Our algorithm relies on existing SMT solver technology so that it can be easily implemented in today's solvers. In its simplest form, the algorithm does not require any modifications to the solver itself but only trivial syntactic modifications to its input. These are: (1) flattening and linearizing the extension axioms; and (2) adding trigger annotations to encode locality constraints for E-matching. In Chapter 5, we evaluate the algorithm on benchmarks generated from heap-manipulating program and discuss further optimizations that can be made in the solver.

## 3.5 Bibliographical note

Sofronie-Stokkermans [54] introduced local theory extensions as a generalization of locality in equational theories [22, 25]. Further generalizations include Psi-local theories [31], which can describe arbitrary theory extensions that admit finite quantifier instantiation. The formalization of our algorithm targets local theory extensions, but

we briefly describe how it can be generalized to handle Psi-locality. The original decision procedure for local theory extensions presented in [54], which is implemented in H-Pilot [32], eagerly generates all instances of extension axioms upfront, before the base theory solver is called. As we show in our experiments, eager instantiation is prohibitively expensive for many local theory extensions that are of interest in verification because it results in a high degree polynomial blowup in the problem size.

In [34], Swen Jacobs proposed an incremental instantiation algorithm for local theory extensions. The algorithm is a variant of model-based quantifier instantiation (MBQI). It uses the base theory solver to incrementally generate partial models from which relevant axiom instances are extracted. The algorithm was implemented as a plug-in to Z3 and experiments showed that it helps to reduce the overall number of axiom instances that need to be considered. However, the benchmarks were artificially generated. Jacob's algorithm is orthogonal to ours as the focus of this paper is on how to use SMT solvers for deciding local theory extensions without adding new substantial functionality to the solvers. A combination with this approach is feasible as we discuss in more detail below.

Other variants of MBQI include its use in the context of finite model finding [50], and the algorithm described in [24], which is implemented in Z3. This algorithm is complete for the so-called almost uninterpreted fragment of first-order logic. While this fragment is not sufficiently expressive for the local theory extensions that appear in our benchmarks, it includes important fragments such as Effectively Propositional Logic (EPR). In fact, we have also experimented with a hybrid approach that uses our E-matching-based algorithm to reduce the benchmarks first to EPR and then solves them with Z3's MBQI algorithm.

E-matching was first described in [44], and since has been implemented in various

SMT solvers [15, 23]. In practice, user-provided *triggers* can be given as hints for finer grained control over quantifier instantiations in these implementations. More recent work [19] has made progress towards formalizing the semantics of triggers for the purposes of specifying decision procedures for a number of theories. A more general but incomplete technique [51] addresses the prohibitively large number of instantiations produced by E-matching by prioritizing instantiations that lead to ground conflicts.

# Chapter 4

# An application: synthesizing commutavity conditions

In the last two chapters, we developed decision procedures with a view of expanding the theories an SMT solver can reason about. In this chapter, we look at an application of SMT. This would give us a flavor of the step of encoding a problem of interest to checking of satisfiability of logic formulas, where an SMT solver fits in a system overall, and some of the challenges.

## 4.1 Problem

Recent decades have seen the development of a variety of paradigms to exploit the opportunity for concurrency in multicore architectures, including parallelizing compilers [53], speculative execution (*e.g.* transactional memory [27]), futures, etc. It has been shown, across all of these domains, that understanding the commutativity of concurrent data-structure operations provides a key avenue to improved performance [14] as well as ease of verification [36, 37].

Intuitively, linearizable data-structure operations that commute can be executed concurrently because their effects don't interfere with each other in a harmful way. When using a (linearizable) HashTable, for example, knowledge that `put(x,'a')` commutes with `get(y)` provided that `x` $\neq$ `y` enables significant parallelization opportunities as both can be performed concurrently.

Commutativity conditions are an important part of the concurrent programming toolkit, but they are tedious to specify manually and require nontrivial and error-prone reasoning. Recent advances have been made on verification of commutativity conditions [35], as well as attempts at synthesis based on random interpretation [2] or dynamic profiling. Thus far, however, generating commutativity conditions automatically has been largely overlooked.

We present the first known technique for automatic refinement of commutativity conditions. We build on a vast body of existing research, extending over the last five decades, on specification and representation of abstract data types (ADTs) in terms of logical $(Pre_m, Post_m)$ specifications [7, 20, 21, 29, 42, 43].

## 4.2 Overview

We describe the algorithm with an example. Consider the Set abstract data type (ADT), whose state consists of a single variable, $S$, that stores an *unordered* collection of *unique* elements. We focus on two operations: `contains(`$x$`)/bool` and `add(`$y$`)/bool` (returns `true` if data structure is modified). Clearly `add` and `contains` commute if they refer to different elements in the set. Another case, which is slightly more subtle, is when both `add` and `contains` refer to the same element $e$, and in the prestate $e \in S$. In this case, in both orders of execution `add` and `contains` leave the

```
REFINE$_n^m(H, \Pi)$ {
if valid($H \Rightarrow m \hat{\bowtie} n$) then
$\varphi$ := $\varphi \vee H$;
else if valid($H \Rightarrow m \hat{\not\bowtie} n$) then
$\tilde{\varphi}$ := $\tilde{\varphi} \vee H$;
else
let $\chi_c, \chi_{nc}$ = counterex. to $\hat{\bowtie}$ and $\hat{\not\bowtie}$ (resp.) in
let $p$ = CHOOSE($H, \Pi, \chi_c, \chi_{nc}$) in
REFINE$_n^m(H \wedge p, \Pi \setminus \{p\})$;
REFINE$_n^m(H \wedge \neg p, \Pi \setminus \{p\})$;
}
main {
$\varphi$ := false;  $\tilde{\varphi}$ := false;
try { REFINE$_n^m$(true, $\Pi$); }
catch (InterruptedException e) { skip; }
return($\varphi, \tilde{\varphi}$);
}
```

Figure 4.1: The refinement algorithm for generating a commutativity condition $\varphi$ and non-commutativity condition $\tilde{\varphi}$ for two methods $m$ and $n$.

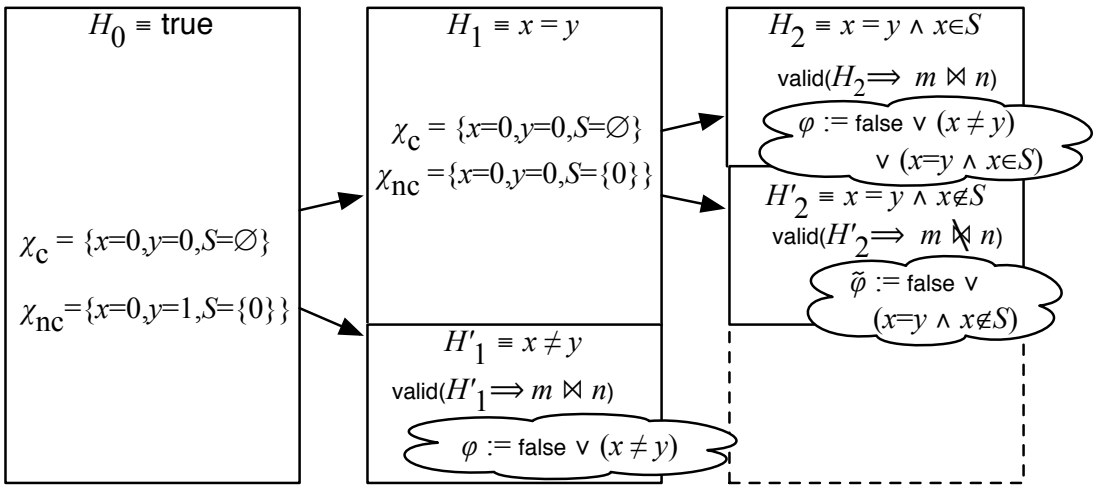



Figure 4.2: An example of how our technique generates commutativity conditions for methods `add` and `contains` operating on a Set. Each subsequent panel depicts a partitioning of the state space. The counterexamples $\chi_c, \chi_{nc}$ give values for the arguments $x$, $y$ and the current state of the set $S$.

set unmodified and return false and true, respectively.

The algorithm we describe in this paper automatically produces a precise logical formula that captures this commutativity condition. It also generates the conditions under which the methods *do not* commute: $x = y \wedge x \notin S$. We explain the algorithm using our running example and, for reference, provide the pseudocode of the algorithm in Figure 4.1.

### 4.2.1 Iterative refinement algorithm

The main thrust of the algorithm is to recursively subdivide the state space via predicates until, at the base case, regions are found that are either entirely commutative or else entirely non-commutative. The conditions we incrementally generate are denoted $\varphi$ and $\tilde{\varphi}$, respectively. In the algorithm, we denote by $H$ the logical formula that describes the current state space at a given recursive call. As expected, we begin with $H_0 =$ true. Our algorithm has three cases for a given $H$: (i) $H$ describes a precondition for $m$ and $n$ in which $m$ and $n$ *always* commute; (ii) $H$ describes a precondition for $m$ and $n$ in which $m$ and $n$ *never* commute; or (iii) neither of the above.

We illustrate how our algorithm proceeds on the running example in Figure 4.2. For these methods, true is visibly not a sound commutativity condition, as our algorithm determines via a *quantifier-free* (QF) *validity query* (described in more detail later) to an SMT solver. This returns the commutativity counterexample described above: $\chi_{\text{c}} = \{x = 0, y = 0, S = \emptyset\}$. true is also not a sufficient precondition for `add` and `contains` *not* to commute. We establish analogously, obtaining the non-commutativity counterexample $\chi_{\text{nc}} = \{x = 0, y = 1, S = \{0\}\}$.

Since $H_0 =$ true is neither a commutativity nor a non-commutativity condition, we must refine $H_0$ into regions (or stronger conditions). In particular, we would like

to perform a *useful* subdivision: Divide $H_0$ into an $H_1$ that allows $\chi_{\text{c}}$ but not $\chi_{\text{nc}}$, and an $H_1'$ that allows $\chi_{\text{nc}}$ but not $\chi_{\text{c}}$. To this end, the choose operation looks for a predicate $p$ (from a suitable set of predicates $\Pi$, discussed later), such that $H_0 \wedge p \Leftarrow \chi_{\text{c}}$ while $H_0 \wedge \neg p \Leftarrow \chi_{\text{nc}}$ (or vice versa). The predicate $x = y$ satisfies this property. In the next two recursive calls, $p$ is added as a conjunct to $H$, as shown in the second column of Figure 4.2: one with $H_1 \equiv \mathsf{true} \wedge x = y$ and one with $H_1' \equiv \mathsf{true} \wedge x \neq y$.

Taking the $H_1'$ case, our algorithm makes another SMT query and finds that $x \neq y$ implies that `add` always commutes with `contains`. At this point, it can update the commutativity condition $\varphi$, letting $\varphi := \varphi \vee H_1'$, adding this $H_1'$ region to the growing disjunction. On the other hand, $H_1$ is neither a sufficient commutativity nor a sufficient non-commutativity condition, and so our algorithm, again, produces the respective counterexamples: $\chi_{\text{c}} = \{x = 0, y = 0, S = \emptyset\}$ and $\chi_{\text{nc}} = \{x = 0, y = 0, S = \{0\}\}$. In this case, our algorithm selects the predicate $x \in S$, and makes two further recursive calls: one with $H_2 \equiv x = y \wedge x \in S$ and another with $H_2' \equiv x = y \wedge x \notin S$. In this case, it finds that $H_2$ is a sufficiently strong precondition for commutativity, while $H_2'$ is a strong enough precondition for non-commutativity. Consequently, $H_2$ is added as a new conjunct to $\varphi$, yielding $\varphi \equiv x \neq y \vee (x = y \wedge x \in S)$. Similarly, $\tilde{\varphi}$ is updated to be: $\tilde{\varphi} \equiv (x = y \wedge x \notin S)$. No further recursive calls are made so the algorithm terminates, and we have obtained a precise (complete) commutativity/non-commutativity specification: $\varphi \vee \tilde{\varphi}$ is valid.

### 4.2.2 Validity query

So far we relied on intuition for when `add` and `contains` commute. We make it this more precise now to give a flavor of the validity queries being generated, and to illustrate how we avoid quantifier alternation which arises when defining commutativity.

Let the abstract states be denoted by $\sigma$, $\sigma'$, $\sigma_m$ etc. (in our example, the abstract state was just the contents of the set). Denote by $\sigma \xrightarrow{m(a)/r} \sigma'$ the predicate which is true iff on application of method $m$ with arguments $a$ on state $\sigma_0$, the return value is $r$ and new state is $\sigma'$. In our example, for `add` this predicate holds when $r = (a \notin S)$ and $S' = S \cup \{a\}$, and for `contains` this predicate holds when $r = (a \in S)$ and $S' = S$. We note that we only work with deterministic systems (this was not a limitation in our experiments), i.e. for any pre-state and method with specific arguments, there is at most one post-state. At times, there might be no valid post state, and it is important to capture this. For example, one cannot pop an empty queue – so $push$ followed by *pop* is possible, but *pop* as the first operation isn't.

Given the above definition, the definition of commutativity for a pair of methods $m$ and $n$ is: (i) for all abstract states $\sigma_0$, whenever $m$ with arguments $x$ (returning value $r_m$) followed by $n$ with arguments $y$ (returning value $r_n$) is possible, then the reverse application $n(y)$ followed by $m(x)$ is also possible and gives the same abstract state and return values (ii) vice-versa. In our notation, it would correspond to checking:

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n.$$
$$\left(\sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \implies \left(\exists \sigma_3 . \sigma_0 \xrightarrow{n(y)/r_n} \sigma_3 \xrightarrow{m(x)/r_m} \sigma_2\right)\right)$$
$$\wedge \left(\sigma_0 \xrightarrow{n(y)/r_n} \sigma_1 \xrightarrow{m(x)/r_m} \sigma_2 \implies \left(\exists \sigma_3 . \sigma_0 \xrightarrow{m(x)/r_m} \sigma_3 \xrightarrow{n(y)/r_n} \sigma_2\right)\right)$$

As one can see above, there is quantifier alternation if we use the natural encoding. When translated to a satisfiability query for an SMT solver, the inner existential quantifier stays as a universal quantifier. Since SMT solvers cannot handle quantifiers very well, we do a transformation which allows us to avoid quantifier alternation. We enforce that there is always a post-state by adding a new $Err$ state in our set of abstract

states. Whenever, there is no post-state for a given state, method and its arguments, we add a transition to the abstract $Err$ state. Once in $Err$ state, we always stay in $Err$ state. Under this modified encoding, it is easy to prove that the following check encodes commutativity defined above:

$$\begin{aligned}
&\forall \sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4, x, y, r_m, r_n, r'_m, r'_n. \\
&\quad \sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \ \wedge \ \sigma_0 \xrightarrow{n(y)/r'_n} \sigma_3 \xrightarrow{m(x)/r'_m} \sigma_4 \\
&\qquad \wedge \ ((\sigma_2 \neq Err \vee \sigma_4 \neq Err) \implies (r_m = r'_m \wedge r_n = r'_n \wedge \sigma_2 = \sigma_4))
\end{aligned}$$

Intuitively, with *exactly* one post state, the universal quantification works as well as the existential one to get a handle on the post state.

### 4.2.3 System overview

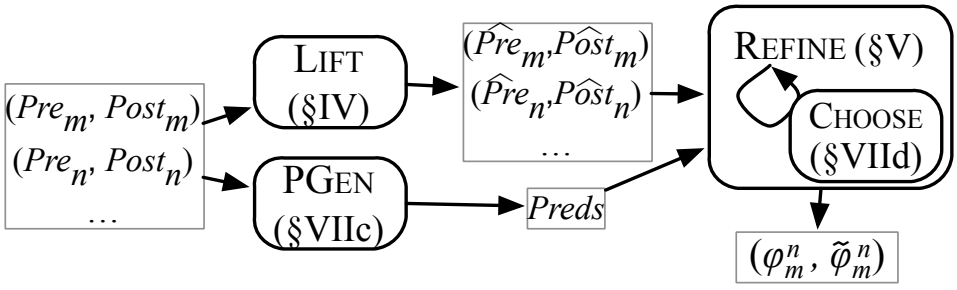
While the algorithm, as outlined so far, executes a relatively standard refinement loop, there are a couple of challenges that are implicit in its description. First, there is the critical question of which predicates to range over during the iterative refinement process. If the predicate vocabulary is not sufficiently expressive, then the algorithm would not be able to converge on precise commutativity and non-commutativity conditions. In Section 4.3, we provide a mechanized solution to this problem, whereby the predicate vocabulary is populated with the atoms that occur in the transition relations' *Pre* and *Post* formulas. As we demonstrate in Section 4.3, this strategy works well in practice. An intuitive explanation is that the *Pre* and *Post* formulas suffice to express the footprint of an operation, and so the atoms comprising them are an effective vocabulary to express when operations do, or do not, interfere.

Having fixed the predicate vocabulary, a second challenge is to prioritize the pred-

icates. This choice essentially drives the iterative refinement loop, and so it controls not only the algorithm's performance, but also the quality (or conciseness) of the resulting conditions. Our choice of next predicate $p$ is governed by two requirements. First, for progress, $p/\neg p$ must eliminate the counterexamples to commutativity/non-commutativity due to the last iteration (where previously selected predicates ensure the same for their respective counterexamples). This may still leave multiple choices, and we propose two heuristics with different trade-offs to break ties. We discuss this in more detail in Section 4.3.

Also, as we discussed in the Section 4.2.2, we would want the validity queries to the solver to be in a fragment for which the solver has complete decision procedures.We refer to the preprocessing of the input described in the section as the LIFT operation.

In summary, the following diagram illustrates the overall flow of our automated algorithm, including the components discussed above to generate useful predicates (PGEN), and choose the next predicate (CHOOSE) within an overall refinement process (REFINE):




In the diagram, we denote the total version of the $Pre_m/Post_m$ specifications (after LIFT preprocessing) as $\widehat{Pre_m}/\widehat{Post_m}$, the generated commutativity condition as $\varphi^n_m$, and the non-commutativity condition $\tilde{\varphi}^n_m$.

## 4.3 Evaluation

We have implemented the algorithm in Section, along with the other parts of the system (illustrated in Section 4.2) in our tool SERVOIS*. We use CVC4 [9] as the backend solver. We evaluated the algorithm on various abstract data structures.In Tables 4.1 and 4.2 and we provide the commutativity conditions generated. The $\varphi_n^m$ column shows the generated commutativity condition. When right moverness ($\triangleright$) conditions are same for a pair of methods, we show them together in one row ($\bowtie$). **Qs** has the number of SMT queries made, and running time in seconds in parentheses. The experiments were run on a 2.53 GHz Intel Core 2 Duo machine with 8 GB RAM.

### 4.3.1 Encoding the transition system

We use an input specification language building on YAML (which has parsers and printers in all common programming languages) with SMTLIB as the logical language. It is human-readable as well as can be easily auto-generated allowing to easily fit in other toolchains [7, 20, 21, 29, 42, 43]. See Appendix A.1.1 for the Counter ADT specification which was derived from the *Pre* and *Post* conditions used in earlier work [35]. The novelty of our work is that we automatically generate commutativity conditions for any implementation that respects these contracts.

The states (state) of a transition system describing an ADT are encoded as list of variables (each as a name, type pair), and each method (methods) specification requires a list of argument types (args), return type (return), and *Pre* (requires) and *Post* (ensures) conditions. The full specifications for Counter, Accumulator, Set, HashTable, and Stack we used can be found in the Appendix A. We used the quantifier-free integer

*http://cs.nyu.edu/~kshitij/projects/servois/

theory in SMTLIB to encode the abstract state and contracts for the counter and accumulator ADTs. For Set, we used the theory of finite sets along with integers to track size; for HashTable we used sets to track the keys, and arrays for the HashMap itself. In order to encode the Stack example, we utilized the observation that for the purpose of pairwise commutativity it is sufficient to track the behavior of boundedly many top elements. In specific, since two operations can *at most* either pop the top two elements or push two elements, tracking four elements is sufficient.

### 4.3.2 Predicate generation (PGEN)

One of the questions that arises is how to obtain a relevant set of the predicates. As mentioned in Section 4.2, our intuition was that the commutativity condition would need to involve terms and predicates that are used to describe the methods. Using this intuition, the procedure we use to generate a set of predicates for input to our REFINE is as follows: (*i*) take all terms that appear in the input specification of the *Pre* and *Post* conditions, grouped by the sort, (*ii*) take all predicate symbols that appear in the specification, and generate all possible atoms that are well-typed using terms extracted. For example, if `size, 1, (size+1)` are terms of sort $\mathbb{Z}$ that appear in the formula along with the predicates = and $\geq$, we generate $(\texttt{size} = 1), (\texttt{size} \geq 1)$, etc. for a total of 18 predicates. We filter out those that are trivial.

By this process, depending on the pair of methods, the number of predicates generated by our implementation of PGEN were (in parenthesis, after filtering): Counter: 25-25 (12-12), Accumulator: 1-20 (0-20), Set: 17-55 (17-34), HashTable: 18-36 (6-36), Stack: 41-61 (41-42).

### 4.3.3 Ranking and picking predicates (CHOOSE)

Even though the number of predicates obtained is relatively small, our algorithm makes two recursive calls at each step. It is thus important to be able to identify *relevant* predicates for the algorithm to be practical.

To this end, in addition to filtering trivial predicates, inspired by CEGAR techniques we prioritize predicates based on the two counterexamples generated from the validity checks in REFINE. Predicates that distinguish between the given counter examples are tried first (call these *distinguishing* predicates). The property of these predicates is that they ensure both counterexamples can be valid on recursing and thus guarantee progress. More formally, CHOOSE must return a predicate such that $\chi_c \Rightarrow H \wedge p$ and $\chi_{nc} \Rightarrow H \wedge \neg p$. In our implementation, we provided the SMT solver all the predicates upfront, which returns the evaluation on the counterexample without any additional queries. This still left us with several predicates, and we discuss the heuristics we tried to break ties.

#### Simple Heuristic

One relatively simple heuristic we tried was to start by picking the predicates with the least number of terms. The intuition was that conditions would at least involve some simple atoms, and would consequently lead to simple conditions. This worked very well, on all our examples this heuristic terminated with precise commutativity conditions. In Tables 4.1 and 4.2, we give the number of queries posed to the solver and total time (in parentheses) consumed by this heuristic.

| Meth. $m(\bar{x})$ | | Meth.$n(\bar{y})$ | Simple | Poke | $\varphi_n^m$ generated by Poke heuristic |
|---|---|---|---|---|---|
| **Counter** | | | Qs (time) | Qs (time) | |
| decrement | $\bowtie$ | decrement | 3 (0.11) | 3 (0.11) | true |
| increment | $\triangleright$ | decrement | 10 (0.36) | 34 (0.91) | $\neg(0 = c)$ |
| decrement | $\triangleright$ | increment | 3 (0.11) | 3 (0.12) | true |
| decrement | $\bowtie$ | reset | 2 (0.10) | 2 (0.10) | false |
| decrement | $\bowtie$ | zero | 6 (0.19) | 26 (0.66) | $\neg(1 = c)$ |
| increment | $\bowtie$ | increment | 3 (0.12) | 3 (0.11) | true |
| increment | $\bowtie$ | reset | 2 (0.09) | 2 (0.10) | false |
| increment | $\bowtie$ | zero | 10 (0.30) | 34 (0.86) | $\neg(0 = c)$ |
| reset | $\bowtie$ | reset | 3 (0.11) | 3 (0.11) | true |
| reset | $\bowtie$ | zero | 9 (0.24) | 30 (0.69) | $0 = c$ |
| zero | $\bowtie$ | zero | 3 (0.11) | 3 (0.11) | true |
| **Accumulator** | | | | | |
| increase | $\bowtie$ | increase | 3 (0.11) | 3 (0.11) | true |
| increase | $\bowtie$ | read | 13 (0.31) | 28 (0.63) | $c + x1 = c$ |
| read | $\bowtie$ | read | 3 (0.09) | 3 (0.09) | true |
| **Set** | | | | | |
| add | $\bowtie$ | add | 10 (0.40) | 140 (4.47) | $[y1 = x1 \wedge y1 \in S] \vee [\neg(y1 = x1)]$ |
| add | $\bowtie$ | contains | 10 (0.42) | 122 (3.63) | $[x1 \in S] \vee [\neg(x1 \in S) \wedge \neg(y1 = x1)]$ |
| add | $\bowtie$ | getsize | 6 (0.21) | 31 (0.93) | $x1 \in S$ |
| add | $\bowtie$ | remove | 6 (0.28) | 66 (2.28) | $\neg(y1 = x1)$ |
| contains | $\bowtie$ | contains | 3 (0.18) | 3 (0.16) | true |
| contains | $\bowtie$ | getsize | 3 (0.13) | 3 (0.13) | true |
| contains | $\bowtie$ | remove | 17 (0.57) | 160 (4.81) | $[S\backslash\{x1\} = \{y1\}] \vee [\neg(S\backslash\{x1\} = \{y1\}) \wedge y1 \in \{x1\} \wedge \ldots] \vee [\ldots]$ |
| getsize | $\bowtie$ | getsize | 3 (0.12) | 3 (0.13) | true |
| getsize | $\bowtie$ | remove | 13 (0.39) | 37 (1.03) | $\neg(y1 \in S)$ |
| remove | $\bowtie$ | remove | 21 (0.75) | 192 (6.47) | $[S\backslash\{y1\} = \{x1\}] \vee [\neg(S\backslash\{y1\} = \{x1\}) \wedge y1 \in \{x1\} \wedge \ldots] \vee [\ldots]$ |

Table 4.1: Automatically generated commutativity conditions.

| Meth. $m(\bar{x})$ | | Meth.$n(\bar{y})$ | Simple | Poke | $\varphi_n^m$ generated by Poke heuristic |
|---|---|---|---|---|---|
| **HashTable** | | | Qs (time) | Qs (time) | |
| get | $\bowtie$ | get | 3 (0.17) | 3 (0.15) | true |
| get | $\bowtie$ | haskey | 3 (0.14) | 3 (0.14) | true |
| put | $\triangleright$ | get | 13 (0.47) | 74 (2.37) | $[\texttt{H[x1=...]} = \texttt{H} \wedge \texttt{y1} \in \texttt{keys}] \vee [\neg(\texttt{H[x1=...]} = \texttt{H}) \wedge \neg(\texttt{y1} = \texttt{x1})]$ |
| get | $\triangleright$ | put | 10 (0.37) | 48 (1.54) | $[\texttt{H[y1]} = \texttt{y2}] \vee [\neg(\texttt{H[y1]} = \texttt{y2}) \wedge \neg(\texttt{y1} = \texttt{x1})]$ |
| remove | $\triangleright$ | get | 3 (0.17) | 3 (0.16) | true |
| get | $\triangleright$ | remove | 13 (0.45) | 40 (1.23) | $\neg(\texttt{y1} = \texttt{x1})$ |
| get | $\bowtie$ | size | 3 (0.14) | 3 (0.14) | true |
| haskey | $\bowtie$ | haskey | 3 (0.14) | 3 (0.14) | true |
| haskey | $\bowtie$ | put | 10 (0.37) | 52 (1.63) | $[\texttt{y1} \in \texttt{keys}] \vee [\neg(\texttt{y1} \in \texttt{keys}) \wedge \neg(\texttt{y1} = \texttt{x1})]$ |
| haskey | $\bowtie$ | remove | 17 (0.59) | 44 (1.36) | $[\texttt{x1} \in \texttt{keys} \wedge \neg(\texttt{y1} = \texttt{x1})] \vee [\neg(\texttt{x1} \in \texttt{keys})]$ |
| haskey | $\bowtie$ | size | 3 (0.14) | 3 (0.14) | true |
| put | $\bowtie$ | put | 24 (0.97) | 357 (13.50) | $[\texttt{H[y1]} = \texttt{y2} \wedge \texttt{x2} = \texttt{H[x1]} \wedge \ldots] \vee [\texttt{H[y1]} = \texttt{y2} \wedge \texttt{x2} = \texttt{H[x1]} \wedge \ldots] \vee [\ldots]$ |
| put | $\bowtie$ | remove | 6 (0.30) | 33 (1.26) | $\neg(\texttt{y1} = \texttt{x1})$ |
| put | $\bowtie$ | size | 6 (0.29) | 23 (0.82) | $\texttt{x1} \in \texttt{keys}$ |
| remove | $\bowtie$ | remove | 21 (0.89) | 192 (6.95) | $[\texttt{keys}\backslash\{\texttt{x1}\} = \{\texttt{y1}\}] \vee [\neg(\texttt{keys}\backslash\{\texttt{x1}\} = \{\texttt{y1}\}) \wedge \texttt{y1} \in \{\texttt{x1}\} \wedge \ldots] \vee [\ldots]$ |
| remove | $\bowtie$ | size | 13 (0.45) | 37 (1.13) | $\neg(\texttt{x1} \in \texttt{keys})$ |
| size | $\bowtie$ | size | 3 (0.14) | 3 (0.14) | true |
| **Stack** | | | | | |
| clear | $\bowtie$ | clear | 3 (0.13) | 3 (0.13) | true |
| clear | $\bowtie$ | pop | 2 (0.10) | 2 (0.11) | false |
| clear | $\bowtie$ | push | 2 (0.12) | 2 (0.11) | false |
| pop | $\bowtie$ | pop | 6 (0.23) | 20 (0.62) | $\texttt{nextToTop} = \texttt{top}$ |
| push | $\triangleright$ | pop | 72 (2.14) | 115 (3.53) | $\neg(0 = \texttt{size}) \wedge \texttt{top} = \texttt{x1}$ |
| pop | $\triangleright$ | push | 34 (0.99) | 76 (2.21) | $\texttt{y1} = \texttt{top}$ |
| push | $\bowtie$ | push | 13 (0.58) | 20 (0.72) | $\texttt{y1} = \texttt{x1}$ |

Table 4.2: Automatically generated commutativity conditions (continued).

#### Poke Heuristic

Though the simple heuristic produces precise conditions, we now focus on the *qualitative* aspect of our synthesis algorithm. We found that in some cases the simple CHOOSE heuristic would pick predicates to split on that could have been technically avoided in the commutativity condition. Not an issue from correctness point of view, nevertheless, we tried a heuristic which tries more aggressively to find *concise* conditions in addition to being precise.

We call this *poke* heuristic, which in order to decide which predicate to pick, recurses one level deep on each predicate and computes number of distinguishing predicates would the two calls have. The sum of values returned by the two calls becomes the weight of the predicate. We then pick the predicate with lowest weight (fewest remaining distinguishing predicates). This heuristic was found to converge much faster to the more relevant predicates. This requires more calls to the SMT solver, but since the queries were relatively simple for CVC4, it was not overall an issue. The conditions in Tables 4.1 and 4.2 are those generated by the Poke heuristic. Please see the appendix for a comparison with those generated by Simple heuristic.

### 4.3.4 Validation

Although our algorithm is sound, we manually validated the implementation of SERVOIS by examining its output and comparing the generated commutativity conditions with those manually written in prior works. In the case of the Accumulator and Counter, our commutativity conditions were identical to those given in [35]. For the Set data-structure, the work of [35] used a less precise Set abstraction, so we instead validated against the conditions of [38]. For the HashTable, we validated that our

conditions matched those given by Dimitrov *et al.* [18].

## 4.4 Conclusion

Our work shows that it possible to automatically generate commutativity conditions, something that was done manually so far. The conditions are correct by construction and ensure special cases aren't missed. These conditions can be derived statically, and used in a variety of contexts including transactional boosting [27], open nested transactions [45], and other non-transactional concurrency paradigms such as race detection [18], automatic parallelization [53], etc.

# Chapter 5

# Implementation and experiments

## 5.1 Theory of finite sets with cardinality

We have implemented a decision procedure based on the rules in Chapter 2 in the SMT solver CVC4. In this section, we describe details related to the implementation, and share experimental results.

### 5.1.1 Proof strategy

The decision procedure can be thought as a specific strategy of applying the rules given in Section 2.2. We recall the categorization of the rules given in 2.3:

- $\mathcal{R}_1$: (necessary) membership predicate reasoning rules
- $\mathcal{R}_2$: (necessary) graph rules to reason about cardinality operator
- $\mathcal{R}_3$: (necessary) rules to handle cardinality constraints imposed by membership reasoning

- $\mathcal{R}_4$: optional propagation and split rules other than Rule 45 (GUESS LOWER BOUND).
- $\mathcal{R}_5$: Rule 45 (GUESS LOWER BOUND).

Our proof strategy can be summarized as follows:

1. Any time a contradiction rule is applicable, apply the rule and close the branch.
2. Apply propagation rules in $\mathcal{R}_1$ and $\mathcal{R}_4$ to saturation.
3. Apply all rules, including split rules, in $\mathcal{R}_1$ and $\mathcal{R}_4$ to saturation.
4. Apply a introduce or merge rule in $\mathcal{R}_2$. After the application, apply rules in $\mathcal{R}_1$, $\mathcal{R}_4$, and Rule 42 (GUESS EMPTY SET) to saturation.
5. Apply rules in $\mathcal{R}_2$ to saturation following above strategy.
6. Apply all rules in calculus, including $\mathcal{R}_3$ and $\mathcal{R}_5$, to saturation.

Note that rules in $\mathcal{R}_1$ do not introduce any new set terms. Further, if there are no constraints of form 7 (see 2.2, Normal form), then we can

### 5.1.2 Data structures

**Reasoning modulo equality.** The rules in Section 2.2.1 reason modulo equality. We use an incremental, congruence closure module to keep all the constraints modulo equality.

**Propagation rules.** In order to handle the propagation rules efficiently, we maintain for each set term all the other terms it appears in. For an incremental procedure, this is important. This helps us avoid looping over all the constraints, and we can detect which propagation rules apply efficiently.

| file | # queries | # sat | # unsat | time (sec.) | # decisions |
|---|---|---|---|---|---|
| deepmeas0 | 138 | 103 | 35 | 0.20 | 967 |
| ListConcat | 178 | 173 | 5 | 0.46 | 2389 |
| ListElem | 38 | 27 | 11 | 0.04 | 182 |
| ListElts | 470 | 358 | 112 | 0.55 | 2682 |
| listSetDemo | 470 | 358 | 112 | 0.58 | 2682 |
| listSet | 393 | 301 | 92 | 0.44 | 2096 |
| meas10 | 216 | 170 | 46 | 0.24 | 1248 |
| meas11 | 33 | 30 | 3 | 0.05 | 184 |
| meas9 | 264 | 210 | 54 | 0.16 | 1064 |
| refinements101reax | 355 | 226 | 129 | 0.18 | 487 |
| SS | 1795 | 801 | 994 | 1.15 | 4214 |
| stacks0 | 189 | 115 | 74 | 0.13 | 643 |
| TalkingAboutSets | 6241 | 4566 | 1675 | 9.33 | 91534 |
| UniqueZipper | 1907 | 1674 | 233 | 6.21 | 26050 |
| zipper0 | 963 | 870 | 93 | 2.46 | 11840 |
| zipper | 8323 | 8017 | 306 | 30.06 | 359863 |

Table 5.1: Performance on benchmarks generated by a static verification tool for Haskell.

### 5.1.3 Experimental results: finite sets

We test our procedure on benchmarks generated by a static verification tool for Haskell*. The constraints do not contain any cardinality constraints. These are incremental (multiple satisfiability checks in one file) benchmarks. The largest benchmark (zipper) has over 8,000 queries (see Table 5.1, # queries column). The next two columns are sat and unsat for number of satisfiable and unsatisfiable queries in the benchmark respectively. The time column is the time taken by the procedure in seconds. Finally, decisions column is the number of decisions (guesses) made by the SAT solver. Both the running time as well as number of decisions are for the whole benchmark (i.e. cumulative over all the queries in a benchmark).

Table 5.1 shows the results for a baseline implementation. On an average, one

*Ranjit Jhala, via the SMT-LIB mailing list, *submitting SMTLIB2 benchmarks?*, Wed Jul 24 11:29:26 EDT 2013

| file | time (sec.) base. | # decisions base. | time (sec.) opt. | # decisions opt. |
|---|---|---|---|---|
| deepmeas0 | 0.20 | 967 | 0.19 | 933 |
| ListConcat | 0.46 | 2389 | 0.42 | 2223 |
| ListElem | 0.04 | 182 | 0.04 | 159 |
| ListElts | 0.55 | 2682 | 0.60 | 2646 |
| listSetDemo | 0.58 | 2682 | 0.54 | 2646 |
| listSet | 0.44 | 2096 | 0.43 | 1977 |
| meas10 | 0.24 | 1248 | 0.25 | 1214 |
| meas11 | 0.05 | 184 | 0.05 | 166 |
| meas9 | 0.16 | 1064 | 0.15 | 985 |
| refinements101reax | 0.18 | 487 | 0.18 | 487 |
| SS | 1.15 | 4214 | 1.38 | 4174 |
| stacks0 | 0.13 | 643 | 0.14 | 643 |
| TalkingAboutSets | 9.33 | 91534 | 9.81 | 91045 |
| UniqueZipper | 6.21 | 26050 | 4.33 | 26050 |
| zipper0 | 2.46 | 11840 | 2.48 | 11840 |
| zipper | 30.06 | 359863 | 28.98 | 359863 |

Table 5.2: Comparison between baseline (base.) and additional use of Rule 29 (opt.).

| file | time (sec.) base. | # decisions base. | time (sec.) opt. | # decisions opt. |
|---|---|---|---|---|
| deepmeas0 | 0.20 | 967 | 0.09 | 265 |
| ListConcat | 0.46 | 2389 | 0.19 | 847 |
| ListElem | 0.04 | 182 | 0.03 | 47 |
| ListElts | 0.55 | 2682 | 0.42 | 860 |
| listSetDemo | 0.58 | 2682 | 0.33 | 860 |
| listSet | 0.44 | 2096 | 0.29 | 478 |
| meas10 | 0.24 | 1248 | 0.11 | 219 |
| meas11 | 0.05 | 184 | 0.03 | 41 |
| meas9 | 0.16 | 1064 | 0.10 | 323 |
| refinements101reax | 0.18 | 487 | 0.12 | 29 |
| SS | 1.15 | 4214 | 1.00 | 1144 |
| stacks0 | 0.13 | 643 | 0.09 | 101 |
| TalkingAboutSets | 9.33 | 91534 | 6.24 | 29959 |
| UniqueZipper | 6.21 | 26050 | 2.90 | 6755 |
| zipper0 | 2.46 | 11840 | 1.44 | 2315 |
| zipper | 30.06 | 359863 | 10.30 | 42896 |

Table 5.3: Comparison of performance between baseline (base.) and when assigning different values by default to shared element variables if they are unconstrained (opt.).

| file | time (sec.) CVC4 | # decisions CVC4 | time (sec.) Z3 | # decisions Z3 |
|---|---|---|---|---|
| deepmeas0 | 0.13 | 162 | 0.04 | 611 |
| ListConcat | 0.15 | 626 | 0.05 | 1399 |
| ListElem | 0.03 | 24 | 0.00 | 85 |
| ListElts | 0.41 | 729 | 0.06 | 1125 |
| listSetDemo | 0.30 | 729 | 0.10 | 1125 |
| listSet | 0.29 | 343 | 0.07 | 665 |
| meas10 | 0.10 | 182 | 0.02 | 348 |
| meas11 | 0.02 | 23 | 0.01 | 96 |
| meas9 | 0.15 | 240 | 0.03 | 422 |
| refinements101reax | 0.13 | 29 | 0.03 | 82 |
| SS | 1.03 | 1001 | 0.15 | 1268 |
| stacks0 | 0.06 | 101 | 0.02 | 404 |
| TalkingAboutSets | 5.74 | 23456 | 6.60 | 63301 |
| UniqueZipper | 2.74 | 6755 | 1.04 | 24078 |
| zipper0 | 1.40 | 2315 | 0.26 | 5441 |
| zipper | 10.05 | 42896 | 3.41 | 64978 |

Table 5.4: Comparison of performance between optimized CVC4 implementation with a translation to Z3 using extension of arrays.

query in a benchmark takes a fraction of a second. We discuss two optimizations we found to be helpful on these benchmarks. First, we found that adding an additional propagation rule, specifically, 29 reduced the number of decisions in some cases. Comparison with baseline results (Table 5.1) is in Table 5.2. Second, if the element variables are shared with another theory, we found that it helps significantly to assign different values by default to variables. Comparison with baseline results (Table 5.1) is in Table 5.3.

Finally, we compare the configuration with both of the optimizations enabled with an encoding of the set operations using Z3's extended array operators. The comparison is in Table 5.4. Both CVC4 as well as Z3 can solve all the queries in all the benchmarks. Z3 is faster on all but one benchmark, though the running times are comparable. Both finish in under a second on benchmarks with less than a thousand

| file | output | time (sec.) | # decisions |
|---|---|---|---|
| card | unsat | 0.00 | 0 |
| card-2 | sat | 0.01 | 14 |
| card-3 | unsat | 0.00 | 0 |
| card-6 | unsat | 0.01 | 0 |
| card-7 | sat | 0.03 | 384 |
| cade07-vc1 | unsat | 0.00 | 4 |
| cade07-vc2 | unsat | 0.01 | 6 |
| cade07-vc2a | unsat | 0.02 | 32 |
| cade07-vc2b | sat | 1.03 | 4730 |
| cade07-vc3 | unsat | 0.00 | 1 |
| cade07-vc3a | unsat | 0.00 | 3 |
| cade07-vc3b | sat | 0.45 | 1305 |
| cade07-vc4 | unsat | 8.82 | 12237 |
| cade07-vc4b | | Timeout | |
| cade07-vc5 | unsat | 50.01 | 20668 |
| cade07-vc5b | | Timeout | |
| cade07-vc6 | unsat | 0.91 | 2193 |
| cade07-vc6a | unsat | 0.37 | 1284 |
| cade07-vc6b | sat | 2.71 | 6420 |
| cade07-vc6c | sat | 1.67 | 5211 |

Table 5.5: Benchmarks involving cardinality reasoning

queries. On benchmarks with more than thousand queries, CVC4 is faster on one benchmark and slower on the other four. Note that on all the benchmarks CVC4 takes fewer decisions. Given the differences in architecture, it is not possible to conclusively say anything, but it warrants further investigation.

### 5.1.4 Experimental results: finite sets and cardinality

We have an initial implementation for cardinality reasoning, built on top of the implementation for theory of finite sets in CVC4. We have tested it on simple hand-crafted benchmarks, and benchmarks derived from [40]. The results are in Table 5.5. Our initial implementation is able to solve 13 of the 15 benchmarks from [40]. The imple-

mentation is a slight variant of the calculus and proof strategy proposed. As future work, we plan a more detailed analysis of an improved implementation on a broader set of benchmarks.

Though we have not re-run the algorithms from [39, 40, 56], we report here the experimental results as stated in the respective papers. As the experiments were run on different machines, the comparison is only indicative. In [40], the algorithm from [39] is reported to solve 12 of the 15 benchmarks with a timeout of 100 seconds. In [40], the algorithm in the paper is reported to solve 11 of the 15 benchmarks with a timeout of 100 seconds. In [56], the algorithm in the paper is reported to solve all 15 benchmarks.

## 5.2 Local theory extensions

We evaluated our techniques for deciding local theory extensions (see Chapter 3) on a set of benchmarks generated by the deductive verification tool GRASShopper [26]. The benchmarks encode memory safety and functional correctness properties of programs that manipulate complex heap-allocated data structures. The programs are written in a type-safe imperative language without garbage collection. The tool makes no simplifying assumptions about these programs like acyclicity of heap structures.

GRASShopper supports mixed specifications in (classical) first-order logic and separation logic (SL) [52]. The tool reduces the program and specification to verification conditions that are encoded in a hierarchical combination of (Psi-)local theory extensions. This hierarchy of extensions is organized as follows:

1. *Base theory:* at the lowest level we have `UFLIA`, the theory of uninterpreted functions and linear integer arithmetic, which is directly supported by SMT

solvers.

2. *GRASS:* the first extension layer consists of the theory of graph reachability and stratified sets. This theory is a disjoint combination of two local theory extensions: the theory of linked lists with reachability [41] and the theory of sets over interpreted elements [59].

3. *Frame axioms:* the second extension layer consists of axioms that encode the frame rule of separation logic. This theory extension includes arrays as a subtheory.

4. *Program-specific extensions:* The final extension layer consists of a combination of local extensions that encode properties specific to the program and data structures under consideration. These include:

   - axioms defining memory footprints of SL specifications,
   - axioms defining structural constraints on the shape of data structures,
   - sorted constraints, and
   - axioms defining partial inverses of certain functions, e.g., to express injectivity of functions and to specify the content of data structures.

We refer the interested reader to [46–48] for further details about the encoding.

The programs considered include sorting algorithms, common data structure operations, such as inserting and removing elements, as well as complex operations on abstract data types. Our selection of data structures consists of singly and doubly-linked lists, sorted lists, nested linked lists with head pointers, binary search trees, skew heaps, and a union find data structure. The input programs comprise 108 procedures with a total of 2000 lines of code, 260 lines of procedure contracts and loop

invariants, and 250 lines of data structure specifications (including some duplicate specifications that could be shared across data structures). The verification of these specifications are reduced by GRASShopper to 816 SMT queries, each serves as one benchmark in our experiments. 802 benchmarks are unsatisfiable. The remaining 14 satisfiable benchmarks stem from programs that have bugs in their implementation or specification. All of these are genuine bugs that users of GRASShopper made while writing the programs.[†] We considered several versions of each benchmark, which we describe in more detail below. Each of these versions is encoded as an SMT-LIB 2 input file.

### 5.2.1 Experimental setup

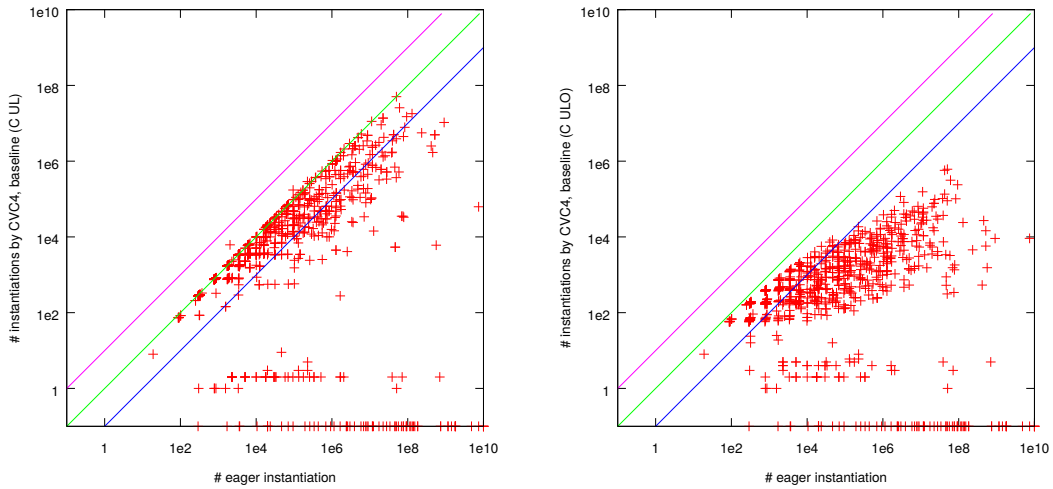
All experiments were conducted on the StarExec platform [55] with a CPU time limit of one hour and a memory limit of 100 GB. We focus on the SMT solvers CVC4 [8] and Z3 [16][‡] as both support `UFLIA` and quantifiers via E-matching. This version of CVC4 is a fork of v1.4 with special support for quantifiers.[§]

In order to be able to test our approach with both CVC4 and Z3, wherever possible we transformed the benchmarks to simulate our algorithm. We describe these transformations in this paragraph. First, the quantified formulas in the benchmarks were linearized and flattened, and annotated with patterns to simulate Step 1(a) of our algorithm (this was done by GRASShopper in our experiments, but may also be handled by an SMT solver aware of local theories). Both CVC4 and Z3 support using these annotations for controlling instantiations in their E-matching procedures. In or-

[†]See `www.cs.nyu.edu/~kshitij/localtheories/` for the programs and benchmarks used.

[‡]We used the version of Z3 downloaded from the git master branch at `http://z3.codeplex.com` on Jan 17, 2015.

[§]This version is available at `www.github.com/kbansal/CVC4/tree/cav14-lte-draft`.




(a) CVC4 with baseline algorithm (b) CVC4 with optimized algorithm

Figure 5.1: # of eager instantiations vs. E-matching instantiations inside the solver

der to handle Psi-local theories, the additional terms required for completeness were provided as dummy assertions, so that these appear as ground terms to the solver. In CVC4, we also made some changes internally so as to treat these assertions specially and apply certain additional optimizations which we describe later in this section.

### 5.2.2 Experiment 1

Our first experiment aims at comparing the effectiveness of eager instantiation versus incremental instantiation up to congruence (as done by E-matching). Figure 5.1 charts the number of eager instantiations versus the number of E-matching instantiations for each query in a logarithmic plot.¶ Points lying on the central line have an equal number of instantiations in both series while points lying on the lower line have ten times as many eager instantiations as E-matching instantiations. (The upper line corresponds to $\frac{1}{10}$.) Most benchmarks require substantially more eager instantia-

¶Figure 5.1 does not include timeouts for CVC4.

| family | # | C # | UD time | C # | UL time | C # | ULO time | Z3 # | UD time | Z3 # | UL time | Z3 # | ULO time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sl lists | 139 | 127 | 70 | 139 | 383 | **139** | **17** | 138 | 1955 | 138 | 1950 | 139 | 68 |
| dl lists | 70 | 66 | 1717 | 70 | 843 | **70** | **33** | 56 | 11375 | 56 | 11358 | 70 | 2555 |
| sl nested | 63 | 63 | 1060 | 63 | 307 | **63** | **13** | 52 | 6999 | 52 | 6982 | 59 | 1992 |
| sls lists | 208 | 181 | 6046 | 204 | 11230 | **208** | **3401** | 182 | 20596 | 182 | 20354 | 207 | 4486 |
| trees | 243 | 229 | 2121 | 228 | 22042 | **239** | **7187** | 183 | 41208 | 183 | 40619 | 236 | 27095 |
| soundness | 79 | 76 | 17 | 79 | 1533 | **79** | **70** | 76 | 7996 | 76 | 8000 | 79 | 336 |
| sat | 14 | - | - | 14 | 670 | **14** | **12** | - | - | 10 | 3964 | 14 | 898 |
| total | 816 | 742 | 11032 | 797 | 37009 | **812** | **10732** | 687 | 90130 | 697 | 93228 | 804 | 37430 |

Table 5.6: Comparison of solvers on uninstantiated benchmarks (time in sec.)

tions. We instrumented GRASShopper to eagerly instantiate all axioms. Subfigure (a) compares upfront instantiations with a baseline implementation of our E-matching algorithm. Points along the $x$-axis required no instantiations in CVC4 to conclude unsat. We have plotted the above charts up to $10^{10}$ instantiations. There were four outlying benchmarks where upfront instantiations had between $10^{10}$ and up to $10^{14}$ instances. E-matching had zero instantiations for all four. Subfigure (b) compares against an optimized version of our algorithm implemented in CVC4. It shows that incremental solving reduces the number of instantiations significantly, often by several orders of magnitude. The details of these optimizations are given later in the section.

### 5.2.3 Experiment 2

Next, we did a more thorough comparison on running times and number of benchmarks solved for *uninstantiated benchmarks*. These results are in Table 5.6. The benchmarks are partitioned according to the types of data structures occurring in the programs from which the benchmarks have been generated. Here, "sl" stands for singly-linked, "dl" for double-linked, and "sls" for sorted singly-linked. The binary search tree, skew heap, and union find benchmarks have all been summarized in the

"trees" row. The row "soundness" contains unsatisfiable benchmarks that come from programs with incorrect code or specifications. These programs manipulate various types of data structures. The actual satisfiable queries that reveal the bugs in these programs are summarized in the "sat" row.

We simulated our algorithm and ran these experiments on both CVC4 (C) and Z3 obtaining similar improvements with both. We ran each with three configurations:

**UD** Default. For comparison purposes, we ran the solvers with default options. CVC4's default solver uses an E-matching based heuristic instantiation procedure, whereas Z3's uses both E-matching and model-based quantifier instantiation (MBQI). For both of the solvers, the default procedures are incomplete for our benchmarks.

**UL** These columns refer to the E-matching based complete procedure for local theory extensions (algorithm in Fig. 3.1).[‖]

**ULO** Doing instantiations inside the solver instead of upfront, opens the room for optimizations wherein one tries some instantiations before others, or reduces the number of instantiations using other heuristics that do not affect completeness. The results in these columns show the effect of all such optimizations.

As noted above, the UL and ULO procedures are both complete, whereas UD is not. This is also reflected in the "sat" row in Table 5.6. Incomplete Instantiation-based procedures cannot hope to answer "sat". A significant improvement can be seen between the UL and ULO columns. The general thrust of the optimizations was to avoid blowup of instantiations by doing ground theory checks on a subset of

[‖] The configuration C UL had one memory out on a benchmark in the tree family.

instantiations. Our intuition is that the theory lemmas learned from these checks eliminate large parts of the search space before we do further instantiations.

For example, we used a heuristic for Psi-local theories inspired from the observation that the axioms involving Psi-terms are needed mostly for completeness, and that we can prove unsatisfiable without instantiating axioms with these terms most of the time. We tried an approach where the instantiations were staged. First, the instantiations were done according to the algorithm in Fig. 3.1 for locality with respect to ground terms from the original query. Only when those were saturated, the instantiations for the auxiliary Psi-terms were generated. We found this to be very helpful. Since this required non-trivial changes inside the solver, we only implemented this optimization in CVC4; but we think that staging instantiations for Psi-local theories is a good strategy in general.

A second optimization, again with the idea of cutting instantiations, was adding assertions in the benchmarks of the form $(a = b) \vee (a \neq b)$ where $a$, $b$ are ground terms. This forces an arbitrary arrangement over the ground terms before the instantiation procedure kicks in. Intuitively, the solver first does checks with many terms equal to each other (and hence fewer instantiations) eliminating as much of the search space as possible. Only when equality or disequality is relevant to the reasoning is the solver forced to instantiate with terms disequal to each other. One may contrast this with ideas being used successfully in the care-graph-based theory combination framework in SMT where one needs to try all possible arrangements of equalities over terms. It has been observed that equality or disequality is sometimes relevant only for a subset of pairs of terms. Whereas in theory combination this idea is used to cut down the number of arrangements that need to be considered, we use it to reduce the number of unnecessary instantiations. We found this really helped CVC4

| family | # | C # | PL time | C # | PLO time | Z3 # | PM time | Z3 # | PL time | Z3 # | PLO time |
|---|---|---|---|---|---|---|---|---|---|---|---|
| sl lists | 139 | 139 | 664 | 139 | 20 | **139** | **9** | 139 | 683 | 139 | 29 |
| dl lists | 70 | 70 | 3352 | 70 | 50 | **70** | **41** | 67 | 12552 | 70 | 423 |
| sl nested | 63 | 63 | 2819 | 63 | 427 | **63** | **182** | 56 | 7068 | 62 | 804 |
| sls lists | 208 | 206 | 14222 | 207 | 3086 | **208** | **37** | 203 | 17245 | 208 | 1954 |
| trees | 243 | 232 | 7185 | 243 | 6558 | **243** | **663** | 222 | 34519 | 242 | 8089 |
| soundness | 79 | 78 | 156 | 79 | 49 | **79** | **23** | 79 | 2781 | 79 | 39 |
| sat | 14 | 14 | 85 | **14** | **22** | 13 | 21 | 12 | 1329 | 14 | 109 |
| total | 816 | 802 | 28484 | **815** | 10213 | **815** | **976** | 778 | 76177 | 814 | 11447 |

Table 5.7: Comparison of solvers on partially instantiated benchmarks (time in sec.)

on many benchmarks.

Another optimization was instantiating special cases of the axioms first by enforcing equalities between variables of the same sort, before doing a full instantiation. We did this for axioms that yield a particularly large number of instances (instantiations growing with the fourth power of the number of ground terms). Again, we believe this could be a good heuristic in general.

### 5.2.4 Experiment 3

Effective propositional Logic (EPR) is the fragment of first order-logic consisting of formulas of the shape $\exists \boldsymbol{x} \forall \boldsymbol{y}.G$ with $G$ quantifier-free and where none of the universally quantified variables $\boldsymbol{y}$ appears below a function symbol in $G$. Theory extensions that fall into EPR are always local. Our third exploration is to see if we can exploit dedicated procedures for this fragment when such fragments occur in the benchmarks. For the EPR fragment, Z3 has a complete decision procedure that uses model-based quantifier instantiation. We therefore implemented a hybrid approach wherein we did upfront partial instantiation to the EPR fragment using E-matching with respect to top-level equalities (as described in our algorithm). The resulting EPR benchmark

is then decided using Z3's MBQI mode. This approach can only be expected to help where there are EPR-like axioms in the benchmarks, and we did have some which were heavier on these. We found that on singly linked list and tree benchmarks this hybrid algorithm significantly outperforms all other solver configurations that we have tried in our experiments. On the other hand, on nested list benchmarks, which make more heavy use of purely equational axioms, this technique does not help compared to only using E-matching because the partial instantiation already yields ground formulas.

The results with our hybrid algorithm are summarized in Column Z3 PM of Table 5.7. Since EPR is a special case of local theories, we also tried our E-matching based algorithm on these benchmarks. We found that the staged instantiation improves performance on these as well. The optimization that help on the uninstantiated benchmarks also work here. These results are summarized in the same table.

Overall, our experiments indicate that there is a lot of potential in the design of quantifier modules to further improve the performance of SMT solvers, and at the same time make them complete on more expressive decidable fragments.

# Chapter 6

# Conclusion

Over the last three decades, starting with breakthroughs in practical tools for checking satisfiability of propositional formulas, the role of general-purpose and fully automated reasoning systems has grown tremendously. SMT solvers have built upon the success in SAT solving, with efficient, sound and complete procedures for fragments of first-order logic – maintaining a fine balance between expressiveness and efficiency. In this thesis, we furthered the scope of SMT solvers by developing decision procedures for additional decidable fragments of first-order logic.

First, we developed a decision procedure for the theory of finite sets and cardinality. We extended a calculus for membership, union, intersection, set difference, and singleton sets to reason about cardinality. We also discussed optimizations and practical aspects of adapting the procedure for an SMT solver.

Second, we show how SMT solvers can be used to obtain complete decision procedures for local theory extensions. We used two SMT solvers to implement this algorithm and conducted an extensive experimental evaluation on benchmarks derived from verification conditions for heap-manipulating programs.

Finally, we discussed new applications that were enabled by these procedures and discussed future directions.

# Appendix A

# Synthesizing commutavity conditions: additional data

## A.1 Data structure specifications and untruncated output

We list the full experimental results from Chapter 4. For each data-structure, we provide the abstract data-structure specification used. We also provide the commutativity conditions synthesized by simple and poke heuristics.

### A.1.1 Counter

```
# Counter data structure's abstract definition

name: counter

state:
  - name: contents
    type: Int

states_equal:
  definition: (= contents_1 contents_2)
```

```yaml
methods:
  - name: increment
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new (+ contents 1))
           (= result true))
    terms:
      Int: [contents, 1, (+ contents 1)]
  - name: decrement
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 1)
    ensures: |
      (and (= contents_new (- contents 1))
           (= result true))
    terms:
      Int: [contents, 1, (- contents 1), 0]
  - name: reset
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new 0)
           (= result true))
    terms:
      Int: [contents, 0]
  - name: zero
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new contents)
           (= result (= contents 0)))
    terms:
      Int: [contents, 0]

predicates:
  - name: "="
    type: [Int, Int]
```

- `decrement` ⋈ `decrement`

  Simple:

  true

  Poke:

true

- `increment` ▷ `decrement`

  Simple:

  [`1 = contents`]

  ∨ [¬(`1 = contents`) ∧ ¬(`0 = contents`)]

  Poke:

  ¬(`0 = contents`)

- `decrement` ▷ `increment`

  Simple:

  true

  Poke:

  true

- `decrement` ⋈ `reset`

  Simple:

  false

  Poke:

  false

- `decrement` ⋈ `zero`

  Simple:

  ¬(`1 = contents`)

  Poke:

  ¬(`1 = contents`)

- `increment` ⋈ `increment`

  Simple:

  true

  Poke:

  true

- `increment` ⋈ `reset`

  Simple:

  false

  Poke:

  false

- `increment` ⋈ `zero`

  Simple:

  [`1 = contents`]

  ∨ [¬(`1 = contents`) ∧ ¬(`0 = contents`)]

  Poke:

  ¬(`0 = contents`)

- `reset` ⋈ `reset`

  Simple:

  true

  Poke:

  true

- `reset` ⋈ `zero`

  Simple:

  ¬(`1 = contents`) ∧ `0 = contents`

  Poke:

  `0 = contents`

- `zero` ⋈ `zero`

  Simple:

  true

  Poke:

  true

### A.1.2 Counter (lifted, auto-generated)

```
methods:
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 0)\n\
    \ (and (= contents_new (+ contents 1))\n     (= result true))\n)\n    (and (not\
    \ err) err_new (not (>= contents 0)\n)))"
  name: increment
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 1
    - (+ contents 1)
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 1)\n\
    \ (and (= contents_new (- contents 1))\n     (= result true))\n)\n    (and (not\
    \ err) err_new (not (>= contents 1)\n)))"
  name: decrement
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 1
    - (- contents 1)
    - 0
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 0)\n\
    \ (and (= contents_new 0)\n     (= result true))\n)\n    (and (not err) err_new\
    \ (not (>= contents 0)\n)))"
  name: reset
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 0
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 0)\n\
    \ (and (= contents_new contents)\n     (= result (= contents 0)))\n)\n    (and\
    \ (not err) err_new (not (>= contents 0)\n)))"
  name: zero
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 0
name: counter
predicates:
- name: '='
  type:
```

```
  - Int
  - Int
state:
- name: contents
  type: Int
- name: err
  type: Bool
states_equal:
  definition: '(or (and err_1 err_2) (and (not err_1) (not err_2)

    (= contents_1 contents_2)

    ))'
```

### A.1.3 Accumulator

```
# Accumulator abstract definition

name: accumulator

state:
  - name: contents
    type: Int

options:


states_equal:
  definition: (= contents_1 contents_2)

methods:
  - name: increase
    args:
      - name: n
        type: Int
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and (= contents_new (+ contents n))
           (= result true))
    terms:
      Int: [$1, contents, (+ contents $1)]
  - name: read
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= contents_new contents)
           (= result contents))
    terms:
      Int: [contents]
```

```
predicates:
  - name: "="
    type: [Int, Int]
```

- `increase` ⋈ `increase`

  Simple:

  true

  Poke:

  true

- `increase` ⋈ `read`

  Simple:

  [`x1 = contents` ∧ `contents + x1 = contents`]

  ∨ [¬(`x1 = contents`) ∧ `contents + x1 = contents`]

  Poke:

  `contents + x1 = contents`

- `read` ⋈ `read`

  Simple:

  true

  Poke:

  true

## A.1.4 Set

```
name: set

preamble: |
  (declare-sort E 0)

state:
  - name: S
    type: (Set E)
  - name: size
    type: Int

states_equal:
  definition: (and (= S_1 S_2) (= size_1 size_2))

methods:
```

```
- name: add
  args:
    - name: v
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (ite (member v S)
         (and (= S_new S)
              (= size_new size)
              (not result))
         (and (= S_new (union S (singleton v)))
              (= size_new (+ size 1))
              result))
  terms:
    E: [$1]
    Int: [size, 1, (+ size 1)]
    (Set E): [S, (singleton $1), (union S (singleton $1))]
- name: remove
  args:
    - name: v
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (ite (member v S)
         (and (= S_new (setminus S (singleton v)))
              (= size_new (- size 1))
              result)
         (and (= S_new S)
              (= size_new size)
              (not result)))
  terms:
    E: [$1]
    Int: [size, 1, (- size 1)]
    (Set E): [S, (singleton $1), (setminus S (singleton $1))]
- name: contains
  args:
    - name: v
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (and (= S_new S)
         (= size_new size)
         (= (member v S) result))
  terms:
    E: [$1]
    Int: [size]
    (Set E): [S, (singleton $1), (setminus S (singleton $1))]
- name: getsize
  args: []
  return:
```

```
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= S_new S)
           (= size_new size)
           (= size result))
    terms:
      Int: [size]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
  - name: "="
    type: [(Set E), (Set E)]
  - name: "member"
    type: [E, (Set E)]
```

- `add` ⋈ `add`

  Simple:

  [y1 = x1 ∧ y1 ∈ S]

  ∨ [¬(y1 = x1)]

  Poke:

  [y1 = x1 ∧ y1 ∈ S]

  ∨ [¬(y1 = x1)]

- `add` ⋈ `contains`

  Simple:

  [y1 = x1 ∧ y1 ∈ S]

  ∨ [¬(y1 = x1)]

  Poke:

  [x1 ∈ S]

  ∨ [¬(x1 ∈ S) ∧ ¬(y1 = x1)]

- `add` ⋈ `getsize`

  Simple:

  x1 ∈ S

  Poke:

  x1 ∈ S

- `add` $\bowtie$ `remove`

  Simple:

  $\neg$(`y1 = x1`)

  Poke:

  $\neg$(`y1 = x1`)

- `contains` $\bowtie$ `contains`

  Simple:

  true

  Poke:

  true

- `contains` $\bowtie$ `getsize`

  Simple:

  true

  Poke:

  true

- `contains` $\bowtie$ `remove`

  Simple:

  [`y1 = x1` $\wedge$ `1 = size` $\wedge$ $\neg$(`y1` $\in$ `S`)]

  $\vee$ [`y1 = x1` $\wedge$ $\neg$(`1 = size`) $\wedge$ $\neg$(`y1` $\in$ `S`)]

  $\vee$ [$\neg$(`y1 = x1`)]

  Poke:

  [`S`$\backslash${`x1`} = {`y1`}]

  $\vee$ [$\neg$(`S`$\backslash${`x1`} = {`y1`}) $\wedge$ `y1` $\in$ {`x1`} $\wedge$ $\neg$(`y1` $\in$ `S`)]

  $\vee$ [$\neg$(`S`$\backslash${`x1`} = {`y1`}) $\wedge$ $\neg$(`y1` $\in$ {`x1`})]

- `getsize` $\bowtie$ `getsize`

  Simple:

  true

  Poke:

  true

- `getsize` ⋈ `remove`

  Simple:

  [1 = size ∧ ¬(y1 ∈ S)]

  ∨ [¬(1 = size) ∧ ¬(y1 ∈ S)]

  Poke:

  ¬(y1 ∈ S)

- `remove` ⋈ `remove`

  Simple:

  [1 = size ∧ y1 = x1 ∧ ¬(y1 ∈ S)]

  ∨ [1 = size ∧ ¬(y1 = x1)]

  ∨ [¬(1 = size) ∧ y1 = x1 ∧ ¬(y1 ∈ S)]

  ∨ [¬(1 = size) ∧ ¬(y1 = x1)]

  Poke:

  [S\{y1} = {x1}]

  ∨ [¬(S\{y1} = {x1}) ∧ y1 ∈ {x1} ∧ ¬(y1 ∈ S)]

  ∨ [¬(S\{y1} = {x1}) ∧ ¬(y1 ∈ {x1})]

### A.1.5 HashTable

```
# Hash table data structure's abstract definition

name: HashTable

preamble: |
  (declare-sort E 0)
  (declare-sort F 0)

state:
  - name: keys
    type: (Set E)
  - name: H
    type: (Array E F)
  - name: size
    type: Int

states_equal:
  definition: |
    (and (= keys_1 keys_2)
         (= H_1 H_2)
         (= size_1 size_2))

methods:
```

```
- name: haskey
  args:
    - name: k0
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (and (= keys_new keys)
         (= H_new H)
         (= size_new size)
         (= (member k0 keys) result)
     )
  terms:
    Int: [size]
    E: [$1]
    (Set E): [keys]
    (Array E F): [H]
- name: remove
  args:
    - name: v
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (ite (member v keys)
         (and (= keys_new (setminus keys (singleton v)))
              (= size_new (- size 1))
              (= H_new H)
             result)
         (and (= keys_new keys)
              (= size_new size)
              (= H_new H)
              (not result)))
  terms:
    Int: [size, 1, (- size 1)]
    E: [$1]
    (Set E): [keys, (singleton $1), (setminus keys (singleton $1))]
    (Array E F): [H]
- name: put
  args:
    - name: k0
      type: E
    - name: v0
      type: F
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (ite (member k0 keys)
         (and (= keys_new keys)
              (= size_new size)
              (ite (= v0 (select H k0))
                   (and (not result)
                        (= H_new H))
```

```
                    (and result
                         (= H_new (store H k0 v0)))))
            (and (= keys_new (insert k0 keys))
                 (= size_new (+ size 1))
                 result
                 (= H_new (store H k0 v0))))
    terms:
      Int: [size, 1, (+ size 1)]
      E: [$1]
      F: [$2, (select H $1), ]
      (Set E): [keys, (insert $1 keys)]
      (Array E F): [H, (store H $1 $2)]
  - name: get
    args:
      - name: k0
        type: E
    return:
      - name: result
        type: F
    requires: |
      (member k0 keys)
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= (select H k0) result)
           )
    terms:
      Int: [size]
      E: [$1]
      F: [(select H $1)]
      (Set E): [keys]
      (Array E F): [H]
  - name: size
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= size result))
    terms:
      Int: [size]
      (Set E): [keys]
      (Array E F): [H]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
  - name: "="
    type: [F, F]
  - name: "="
    type: [(Set E), (Set E)]
  - name: "="
    type: [(Array E F), (Array E F)]
  - name: "member"
```

```
type: [E, (Set E)]
```

- `get` ⋈ `get`

  Simple:

  true

  Poke:

  true

- `get` ⋈ `haskey`

  Simple:

  true

  Poke:

  true

- `put` ▷ `get`

  Simple:

  [x2 = H[y1] ∧ y1 ∈ keys]

  ∨ [¬(x2 = H[y1]) ∧ ¬(y1 = x1)]

  Poke:

  [H[x1↦x2] = H ∧ y1 ∈ keys]

  ∨ [¬(H[x1↦x2] = H) ∧ ¬(y1 = x1)]

- `get` ▷ `put`

  Simple:

  [H[y1] = y2]

  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]

  Poke:

  [H[y1] = y2]

  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]

- `remove` ▷ `get`

  Simple:

  true

  Poke:

  true

- `get` ▷ `remove`

  Simple:

  [`1 = size` ∧ ¬(`y1 = x1`)]

  ∨ [¬(`1 = size`) ∧ ¬(`y1 = x1`)]

  Poke:

  ¬(`y1 = x1`)

- `get` ⋈ `size`

  Simple:

  true

  Poke:

  true

- `haskey` ⋈ `haskey`

  Simple:

  true

  Poke:

  true

- `haskey` ⋈ `put`

  Simple:

  [`y1 = x1` ∧ `y1` ∈ `keys`]

  ∨ [¬(`y1 = x1`)]

  Poke:

  [`y1` ∈ `keys`]

  ∨ [¬(`y1` ∈ `keys`) ∧ ¬(`y1 = x1`)]

- haskey $\bowtie$ remove

  Simple:

  $[\texttt{y1} = \texttt{x1} \wedge 1 = \texttt{size} \wedge \neg(\texttt{y1} \in \texttt{keys})]$

  $\vee\, [\texttt{y1} = \texttt{x1} \wedge \neg(1 = \texttt{size}) \wedge \neg(\texttt{y1} \in \texttt{keys})]$

  $\vee\, [\neg(\texttt{y1} = \texttt{x1})]$

  Poke:

  $[\texttt{x1} \in \texttt{keys} \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\neg(\texttt{x1} \in \texttt{keys})]$

- haskey $\bowtie$ size

  Simple:

  true

  Poke:

  true

- put $\bowtie$ put

  Simple:

  $[\texttt{x2} = \texttt{y2} \wedge \texttt{x2} = \texttt{H}[\texttt{y1}] \wedge \texttt{y1} \in \texttt{keys}]$

  $\vee\, [\texttt{x2} = \texttt{y2} \wedge \texttt{x2} = \texttt{H}[\texttt{y1}] \wedge \neg(\texttt{y1} \in \texttt{keys}) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\texttt{x2} = \texttt{y2} \wedge \neg(\texttt{x2} = \texttt{H}[\texttt{y1}]) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\neg(\texttt{x2} = \texttt{y2}) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  Poke:

  $[\texttt{H}[\texttt{y1}] = \texttt{y2} \wedge \texttt{x2} = \texttt{H}[\texttt{x1}] \wedge \texttt{size} + 1 = 1 \wedge \texttt{y1} \in \texttt{keys}]$

  $\vee\, [\texttt{H}[\texttt{y1}] = \texttt{y2} \wedge \texttt{x2} = \texttt{H}[\texttt{x1}] \wedge \texttt{size} + 1 = 1 \wedge \neg(\texttt{y1} \in \texttt{keys}) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\texttt{H}[\texttt{y1}] = \texttt{y2} \wedge \texttt{x2} = \texttt{H}[\texttt{x1}] \wedge \neg(\texttt{size} + 1 = 1) \wedge \texttt{x1} \in \texttt{keys}]$

  $\vee\, [\texttt{H}[\texttt{y1}] = \texttt{y2} \wedge \texttt{x2} = \texttt{H}[\texttt{x1}] \wedge \neg(\texttt{size} + 1 = 1) \wedge \neg(\texttt{x1} \in \texttt{keys}) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\texttt{H}[\texttt{y1}] = \texttt{y2} \wedge \neg(\texttt{x2} = \texttt{H}[\texttt{x1}]) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\neg(\texttt{H}[\texttt{y1}] = \texttt{y2}) \wedge \neg(\texttt{y1} = \texttt{x1})]$

- put $\bowtie$ remove

  Simple:

  $\neg(\texttt{y1} = \texttt{x1})$

Poke:

$\neg(\texttt{y1} = \texttt{x1})$

- $\texttt{put} \bowtie \texttt{size}$

  Simple:

  $\texttt{x1} \in \texttt{keys}$

  Poke:

  $\texttt{x1} \in \texttt{keys}$

- $\texttt{remove} \bowtie \texttt{remove}$

  Simple:

  $[\texttt{1} = \texttt{size} \wedge \texttt{y1} = \texttt{x1} \wedge \neg(\texttt{y1} \in \texttt{keys})]$

  $\vee\, [\texttt{1} = \texttt{size} \wedge \neg(\texttt{y1} = \texttt{x1})]$

  $\vee\, [\neg(\texttt{1} = \texttt{size}) \wedge \texttt{y1} = \texttt{x1} \wedge \neg(\texttt{y1} \in \texttt{keys})]$

  $\vee\, [\neg(\texttt{1} = \texttt{size}) \wedge \neg(\texttt{y1} = \texttt{x1})]$

  Poke:

  $[\texttt{keys} \setminus \{\texttt{x1}\} = \{\texttt{y1}\}]$

  $\vee\, [\neg(\texttt{keys} \setminus \{\texttt{x1}\} = \{\texttt{y1}\}) \wedge \texttt{y1} \in \{\texttt{x1}\} \wedge \neg(\texttt{y1} \in \texttt{keys})]$

  $\vee\, [\neg(\texttt{keys} \setminus \{\texttt{x1}\} = \{\texttt{y1}\}) \wedge \neg(\texttt{y1} \in \{\texttt{x1}\})]$

- $\texttt{remove} \bowtie \texttt{size}$

  Simple:

  $[\texttt{1} = \texttt{size} \wedge \neg(\texttt{x1} \in \texttt{keys})]$

  $\vee\, [\neg(\texttt{1} = \texttt{size}) \wedge \neg(\texttt{x1} \in \texttt{keys})]$

  Poke:

  $\neg(\texttt{x1} \in \texttt{keys})$

- $\texttt{size} \bowtie \texttt{size}$

  Simple:

  true

  Poke:

  true

### A.1.6 Stack

```
# Stack definition

name: stack

preamble: |
  (declare-sort E 0)

state:
  - name: size
    type: Int
  - name: top
    type: E
  - name: nextToTop
    type: E
  - name: secondToTop
    type: E
  - name: thirdToTop
    type: E

states_equal:
  definition:
    (and (= size_1 size_2)
         (or (= size_1 0)
             (and (= size_1 1) (= top_1 top_2))
             (and (= top_1 top_2) (= nextToTop_1 nextToTop_2))))

methods:
  - name: push
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      (>= size 0)
    ensures: |
      (and (= size_new (+ size 1))
           (= top_new v)
           (= nextToTop_new top)
           (= secondToTop_new nextToTop)
           (= thirdToTop_new secondToTop)
           (= result true))
    terms:
      Int: [size, 1, (+ size 1)]
      E: [top, nextToTop, secondToTop, thirdToTop, $1]
  - name: pop
    args: []
    return:
      - name: result
        type: E
    requires: |
      (>= size 1)
    ensures: |
      (and (= size_new (- size 1))
           (= result top)
           (= top_new nextToTop)
           (= nextToTop_new secondToTop)
           (= secondToTop_new thirdToTop))
```

```
    terms:
      Int: [size, 1, (- size 1), 0]
      E: [top, nextToTop, secondToTop, thirdToTop]
  - name: clear
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= size 0)
    ensures: |
      (and (= size_new 0)
           (= result true))
    terms:
      Int: [size, 0]
      E: [top, nextToTop, secondToTop, thirdToTop]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
```

- `clear` ⋈ `clear`

  Simple:

  true

  Poke:

  true

- `clear` ⋈ `pop`

  Simple:

  false

  Poke:

  false

- `clear` ⋈ `push`

  Simple:

  false

  Poke:

  false

- `pop` ⋈ `pop`

  Simple:

nextToTop = top

Poke:

nextToTop = top

- push ▷ pop

  Simple:

  [1 = size ∧ nextToTop = top ∧ nextToTop = thirdToTop ∧ nextToTop = x1]

  ∨ [1 = size ∧ nextToTop = top ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = x1]

  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ top = x1]

  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ top = x1]

  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ top = x1]

  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ top = x1]

  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ top = x1]

  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ top = x1]

  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ top = x1]

  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ top = x1]

  Poke:

  ¬(0 = size) ∧ top = x1

- pop ▷ push

  Simple:

  [nextToTop = y1 ∧ nextToTop = top]

  ∨ [¬(nextToTop = y1) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ y1 = top]

  ∨ [¬(nextToTop = y1) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ y1 = top]

  ∨ [¬(nextToTop = y1) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ y1 = top]

  ∨ [¬(nextToTop = y1) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ y1 = top]

  Poke:

  y1 = top

- `push` $\bowtie$ `push`

  Simple:

  $[\texttt{thirdToTop} = \texttt{y1} \wedge \texttt{thirdToTop} = \texttt{x1}]$

  $\vee\, [\neg(\texttt{thirdToTop} = \texttt{y1}) \wedge \texttt{y1} = \texttt{x1}]$

  Poke:

  $\texttt{y1} = \texttt{x1}$

# Bibliography


[1] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Decision procedures for flat array properties. In *TACAS*, volume 8413 of *LNCS*, pages 15–30. Springer, 2014.

[2] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 241–252. ACM, 2009.

[3] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[4] Kshitij Bansal, Clark Barrett, Andrew Reynolds, and Cesare Tinelli. Decision procedure for finite sets and cardinality in SMT (under preparation).

[5] Kshitij Bansal, Eric Koskinen, and Omer Trip. Commutativity condition refinement (under submission).

[6] Kshitij Bansal, Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies. Deciding local theory extensions via e-matching. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, 2015. San Francisco, USA.

[7] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, 2005.

[8] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *CAV*, pages 171–177, 2011.

[9] 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, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806, pages 171–177. Springer, July 2011.

[10] David A. Basin and Harald Ganzinger. Complexity analysis based on ordered resolution. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 456–465. IEEE, 1996.

[11] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

[12] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[13] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. *J. Autom. Reasoning*, 47(4):341–367, 2011.

[14] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10, 2015.

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

[16] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[17] Leonardo Mendonça de Moura and Grant Olney Passmore. The strategy challenge in SMT solving. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, pages 15–44, 2013.

[18] Dimitar Dimitrov, Veselin Raychev, Martin T. Vechev, and Eric Koskinen. Commutativity race detection. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 33. ACM, 2014.

[19] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Pascal Fontaine and Amit Goel, editors, *SMT 2012*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2013.

[20] George W. Ernst and William F. Ogden. Specification of abstract data types in modula. *ACM Trans. Program. Lang. Syst.*, 2(4):522–543, October 1980.

[21] L. Flon and J. Misra. A unified approach to the specification and verification of abstract data types. In *Proc. Specifications of Reliable Software Conf., IEEE Computer Society*, 1979.

[22] H. Ganzinger. Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 81–90, 2001.

[23] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122, February 2009.

[24] Yeting Ge and Leonardo Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 306–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Robert Givan and David A. McAllester. New results on local inference relations. In *KR*, pages 403–412. Morgan Kaufmann, 1992.

[26] GRASShopper tool web page. `http://cs.nyu.edu/wies/software/grasshopper`. Accessed: Feb 2015.

[27] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'08)*, 2008.

[28] David Hilbert and Wilhelm Ackermann. *Grundzügeder theoretischen Logik (Principles of Mathematical Logic)*, chapter 13, page 112. Springer-Verlag, 1950.

[29] C. A. R. Hoare. Software pioneers. In Manfred Broy and Ernst Denert, editors, *Software Pioneers*, chapter Proof of Correctness of Data Representations, pages 385–396. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[30] Matthias Horbach and Viorica Sofronie-Stokkermans. Obtaining finite local theory axiomatizations via saturation. In *FroCoS*, volume 8152 of *LNCS*, pages 198–213. Springer, 2013.

[31] Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In *TACAS*, pages 265–281, 2008.

[32] Carsten Ihlemann and Viorica Sofronie-Stokkermans. System description: H-pilot. In *CADE*, volume 5663 of *LNCS*, pages 131–139. Springer, 2009.

[33] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. Modular reasoning about heap paths via effectively propositional formulas. In *POPL*, pages 385–396. ACM, 2014.

[34] Swen Jacobs. Incremental instance generation in local reasoning. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 368–382, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Deokhwan Kim and Martin C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 528–541. ACM, 2011.

[36] Eric Koskinen and Matthew J. Parkinson. The push/pull model of transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, Portland, OR, USA, June, 2015*, 2015.

[37] Eric Koskinen, Matthew J. Parkinson, and Maurice Herlihy. Coarse-grained transactions. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 19–30. ACM, 2010.

[38] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 542–555. ACM, 2011.

[39] Viktor Kuncak, HuuHai Nguyen, and Martin Rinard. Deciding boolean algebra with presburger arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, 2006.

[40] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *Conference on Automateded Deduction (CADE-21)*, volume 4603 of *LNCS*, 2007.

[41] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, pages 171–182, 2008.

[42] K. Rustan M. Leino. Specifying and verifying programs in spec#. In *Proceedings of the 6th International Perspectives of Systems Informatics, Andrei Ershov Memorial Conference, PSI 2006*, page 20, 2006.

[43] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[44] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

[45] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007*, pages 68–78. ACM, 2007.

[46] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic Using SMT. In *CAV*, volume 8044 of *LNCS*, pages 773–789. Springer, 2013.

[47] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In *CAV*, volume 3855 of *LNCS*, pages 711–728. Springer, 2014.

[48] Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper: Complete Heap Verification with Mixed Specifications. In *TACAS*. Springer, 2014.

[49] Zvonimir Rakamaric, Jesse D. Bingham, and Alan J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI*, volume 4349 of *LNCS*, pages 106–121. Springer, 2007.

[50] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in

SMT. In M. P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction (Lake Placid, NY, USA)*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.

[51] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2014.

[52] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[53] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, November 1997.

[54] Viorica Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *CADE-20*, volume 3632 of *LNCS*, pages 219–234. Springer, 2005.

[55] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: a Cross-Community Infrastructure for Logic Solving. In *IJCAR*, pages 367–373, 2014.

[56] Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.

[57] Calogero G. Zarba. Combining multisets with integers. In *CADE-18*, 2002.

[58] Calogero G. Zarba. Combining sets with integers. In *Frontiers of Combining Systems, 4th International Workshop, FroCoS 2002*, pages 103–116, 2002.

[59] Calogero G. Zarba. Combining sets with elements. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 762–782. Springer, 2003.