Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic

by

Tim King

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

September 2014

Clark Barrett

Acknowledgements

I would like to thank my advisor Clark Barrett. Without his support, this thesis would not have been possible. I would also like to thank the other members of my reading committee: Thomas Wies and Margaret Wright. Throughout this process, I have been exceptionally fortunate to have been guided and advised by a large number of talented researchers, including David Dill, Bruno Dutertre, Mykel Kochenderfer, Jeremy Levitt, Kedar Namjoshi, Sam Owre, Amir Pnueli, Natarajan Shankar, David Spencer, and Cesare Tinelli. This work owes a great deal of debt to the other CVC4 developers: Kshitij Bansal, Morgan Deters, Dejan Jovanović, and Andrew Reynolds. Daniel Schwartz-Narbonne helped me to edit this thesis. I am thankful for my friends and family for their belief in me and their encouragement. Lastly, I would like to thank Liana Hadarean for undertaking the labor of editing my first drafts and her support in helping me past my many dour moods while writing this. Thank you all.

Abstract

A core technique of modern tools for formally reasoning about computing systems is generating and dispatching queries to automated theorem provers, including Satisfiability Modulo Theories (SMT) provers. SMT provers aim at the tight integration of decision procedures for propositional satisfiability and decision procedures for fixed first-order theories – known as theory solvers. This thesis presents several advancements in the design and implementation of theory solvers for quantifier-free linear real, integer, and mixed integer and real arithmetic. These are implemented within the SMT system CVC4. We begin by formally describing the Satisfiability Modulo Theories problem and the role of theory solvers within CVC4. We discuss known techniques for building solvers for quantifier-free linear real, integer, and mixed integer and real arithmetic around the Simplex for SMT algorithm. We give several small improvements to theory solvers using this algorithm and describe the implementation and theory of this algorithm in detail. To extend the class of problems that the theory solver can robustly support, we borrow and adapt several techniques from linear programming (LP) and mixed integer programming (MIP) solvers which come from the tradition of optimization. We propose a new decision procedure for quantifier-free linear real arithmetic that replaces the Simplex for SMT algorithm with a variant of the Simplex algorithm that performs a form of optimization – minimizing the sum of infeasibilties. In this thesis, we additionally describe techniques for leveraging LP and MIP solvers to improve the performance of SMT solvers without compromising correctness. Previous efforts to leverage such solvers in the context of SMT have concluded that in addition to being potentially unsound, such solvers are too heavyweight to compete in the context of SMT. We present an empirical comparison against other state-of-the-art SMT tools to demonstrate the effectiveness of the proposed solutions.

Contents

Acknowledgements iii
Abstract iv
List of Figures vii
List of Tables ix
Introduction 1

1 Satisfiability Modulo Theories 6
1.1 First-Order Logic 6
1.2 The SMT Problem 10
1.3 Decision Procedures for SAT 14
1.4 DPLL($\mathcal{T}$) 18
1.5 The CVC4 Architecture 19

2 Simplex for Satisfiability Modulo Theories 34
2.1 Abstract Description 35
2.2 Refining the Abstract Algorithm 48

3 Simplex with Sum of Infeasibilities for SMT 84
3.1 Naive Primal Simplex 85
3.2 Sum of Infeasibility 89
3.3 Sum of Infeasibilities Simplex 91
3.4 Experimental Results 101

4 Extending Simplex to Integer and Integer Real Arithmetic 104
4.1 Quantifier-Free Linear Integer Arithmetic 105
4.2 Quantifier-Free Linear Integer Real Arithmetic 106
4.3 Normalization and Simplification 106
4.4 Theory Solvers for `QF_LIRA` 109

5 Leveraging Linear Programming Solvers 119
5.1 Leveraging LP Solvers 120
5.2 Leveraging MIP Solvers 125
5.3 Experiments 133

5.4 Future Work 136

6 Conclusions **142**

A Discussion and Proofs of Miscellaneous Theorems **144**

A.1 Discussion 144

A.2 Proofs of Miscellaneous Theorems 145

Bibliography **165**

List of Figures

1.1 DPLL procedure . . . 17
1.2 Abstraction of the CVC4 SmtEngine . . . 21
1.3 CVC4 SMTCHECKSAT procedure . . . 22
1.4 DPLL($\mathcal{T}$) communication in CVC4 . . . 24
1.5 DPLL($\mathcal{T}$) extended CDCL loop . . . 27
1.6 Abstraction of a theory solver . . . 28
1.7 Example rewrite rules . . . 31

2.1 Procedure for updating the assignment of a non-basic variable . . 39
2.2 An abstract pivot procedure . . . 40
2.3 An abstract update and pivot procedure . . . 41
2.4 Procedure for asserting an upper bound . . . 42
2.5 Procedure for asserting a lower bound . . . 42
2.6 Abstract main loop for SIMPLEXFORSMT theory check . . . 44
2.7 An abstract pivot and update selection routine . . . 45
2.8 Procedure for incrementally processing assertions . . . 65
2.9 Row addition procedure . . . 67
2.10 Pivot procedure with sign changes . . . 67
2.11 Updating a bound count in response to sign changes . . . 72
2.12 Updating a bound count in response to status changes . . . 72
2.13 An update procedure with variable status changes . . . 73
2.14 A check for whether a basic variable in a list is in conflict . . . 74
2.15 Lazy splitting for disequalities . . . 77
2.16 A pivot and update selection routine with heuristic variable orders 79
2.17 Refined SIMPLEXFORSMT check main loop . . . 80
2.18 Pivot and update with sign and status changes . . . 80
2.19 Complete SIMPLEXFORSMT theory check procedure . . . 80

3.1 Primal Simplex procedure . . . 86
3.2 Pivot and update selection routine for primal Simplex . . . 86
3.3 Check procedure for SOISIMPLEX . . . 91
3.4 Selection rules for SOISIMPLEX . . . 92

3.5 Example of violated variables after an update 93
3.6 Conflict minimization routine for SOISimplex 98
3.7 Comparing running times of SimplexForSMT and SOISimplex 102

4.1 Example feasible region 116

5.1 BalancedSolve procedure for linear real arithmetic 121
5.2 ImportAssignment procedure 123
5.3 ExactReseed procedure 124
5.4 IntegerSolve procedure for linear integer arithmetic 126
5.5 Derivation rules for branch-and-cut executions 127
5.6 Replay procedure 129
5.7 The ReplayConflict procedure 130

List of Tables

3.1 Number of pivots performed per call to SOISIMPLEX and SIMPLEXFORSMT 103

5.1 Experimental results on `QF_LRA` and `QF_LIA` benchmarks. 140
5.2 Success rate of reproducing results of MIPSOLVE 141

Introduction

The *Satisfiability Modulo Theories* (SMT) problem is to decide whether or not a logical statement can be made true–or satisfied–within a first-order theory. This allows for asking queries of the form – can you make the statement "x plus y is strictly less than 3, y is not less than 0, and x is equal to 4" true if x and y are real numbers? The intuitive answer is of course not! The role of the theories is to restrict the meanings of statements such as "plus," "greater than" or even "4" so that they match our intuition about arithmetic and formally allow solvers of SMT queries to also conclude this example is unsatisfiable. Solvers for SMT have specialized decision procedures in order to be able to decide individual theories. The focus of this thesis is on the decision procedure for the theories of real, integer, and mixed integer and real arithmetic. SMT solvers aim to be an efficient form of *automated theorem proving* capable of handling computationally challenging problems and are designed to be quick and robust on problems of interest.

SMT solvers were developed to be a back-end technology to perform symbolic reasoning for other tools arising from *formal methods*. These tools automatically generate SMT queries in order to express and reason about computing systems. Computing systems are ubiquitous in modern life. From general purpose computing devices like cellphones to small specialized devices such as thermostats, more and more objects in everyday life contain some form of computational reasoning. As these increasingly complicated devices spread, so too does buggy software and hardware. The common practice used to ameliorate bugs is to test and simulate computing systems to find the presence of bad behaviors. These practices have the advantage of being easy to understand. They are also effective at catching many simple bugs and demonstrating that the program or circuit can perform as expected on many examples. However, a problem with testing and simulation is that while they can prove the existence of bugs, they typically cannot guarantee the non-existence of bugs due to the large number of possible behaviors.

The subfield of formal methods attempts to use computational tools to logically specify and reason about computational systems. Techniques such as model checking and abstract interpretation can be used to prove the absence of bugs in existing software systems while areas such as synthesis attempt to automatically build systems that by construction must fulfill a given specification. There are also tools to automatically generate test cases that are guaranteed to exhibit new traces for every generated test. What unites formal methods is the use of symbolic, formal and logical specifications for computers and computer programs, as well as the development and application of computational tools to assist in reasoning about such logical specifications.

The improvements in model checking algorithms over the past 30 years have closely followed improvements in symbolic reasoning. Model checkers began using explicit constructions of finite automata [30]. A great advance in making model checking scalable was the creation of symbolic model checking to represent sets of states and the semantics of transition relations using a logical representation called Binary Decision Diagrams (BDDs) [62,83]. While BDD based model checking continues to have its strengths, most state-of-the-art research in model checking and verification is based upon using propositional satisfiability (SAT) queries as a way of modeling the executions of finite state systems [15]. What has driven the success of this approach is the rapid improvement in SAT solvers over the past two decades. The growth of efficient-in-practice SAT solvers has led to these solvers being used in a wide variety of new and unforeseen ways, including solving mathematical conjectures [75]. Some have even called it the "SAT Revolution" [25].

SMT solvers lift propositional satisfiability to reason over first-order logical theories. SMT solvers are built by tightly interleaving the steps of a modern SAT solver with calls to theory solvers that reason over finite sets of first-order literals for built-in theories. The built-in theories give semantics or meaning to such first-order statements, and these semantics allow us to conclude that "x must be strictly less than 3 if x plus y is strictly less than 3, and y is not less than 0." By extending the language the solvers can natively handle, SMT solvers offer a richer and more natural language for expressing a multitude of problems. From describing the temperature gauges in a controller to modeling computer memory to modeling the relative positions of airplanes in air traffic control systems, the list of applications of SMT solvers keeps growing [8].

This thesis is concerned with improving the theory solvers for the satisfiability of quantifier-free linear real arithmetic, quantifier-free linear integer arithmetic, and quantifier-free linear mixed integer and real arithmetic problems. This thesis makes the following contributions:

1. The thesis discusses in detail the design and construction of theory solvers for arithmetic. Such a discussion is usually considered too low-level for research papers in SMT. This leads to an unfortunate situation where the implementers of SMT solvers must independently learn the small pitfalls and optimization improvements. This thesis attempts to address all of the major design decisions and procedures underlying CVC4's state-of-the-art theory solver for arithmetic. We additionally describe key data structures and formalize the internally used inference rules. This discussion is given mostly in terms of the Simplex for SMT algorithm as it forms the core of this work and is the simplest decision procedure described.

2. The thesis additionally gives a number of small improvements to the Sim-

plex for SMT algorithm. This includes a method of bookkeeping for efficiently detecting theory conflicts. We describe a new method of strengthening the conflicts found by the Simplex for SMT algorithm. A new model generation procedure is described that allows the decision procedure to handle disequalities via splitting-on-demand lemmas.

3. A new theory solver is given for quantifier-free linear real arithmetic that extends the core infrastructure for the Simplex for SMT algorithm and replaces the central decision procedure. This new decision procedure is a variant of the Simplex algorithm, which on every step minimizes the sum-of-infeasibilities function. By adding this minimization, the algorithm seeks to address a major problem with the robustness of the Simplex for SMT procedure. We give experimental results that compare the two algorithms and provide partial insights into the strengths of both. A new heuristic conflict minimization procedure for this method is given that avoids performing additional search.

4. A major contribution of this thesis is a new method for taking advantage of existing floating point linear programming Simplex solvers and branch-and-cut solvers. Because floating point arithmetic is inexact, rounding errors can lead to incorrect results, making inexact solvers inappropriate for direct use in theorem proving. Previous efforts to leverage such solvers in the context of SMT have concluded that in addition to being potentially unsound, such solvers are too heavyweight to compete in the context of SMT. We describe techniques for integrating Linear Programming solvers and branch-and-cut Mixed Integer Programming solvers that can dramatically improve the performance of SMT solvers on challenging instances without compromising correctness. The solution leaves the search of the Mixed Integer Programming solver unchanged, but requires the solver to implement additional logging while generating cutting planes.

Organization of The Thesis

Chapter 1 begins this thesis with an introduction to the theoretical and practical background of SMT solving. It gives a brief introduction to many-sorted first-order logic, the fundamental language of SMT solvers. The chapter then formally defines the SMT problem. The SAT problem is defined as a subcase of SMT. We then give an introduction to three procedures for solving SAT: the classical Davis-Putnam and Davis-Putnam-Logemann-Loveland (DPLL) procedures and their modern synthesis in the Conflict Driven Clause Learning (CDCL) procedure. We move from SAT, to discussing the DPLL($\mathcal{T}$) architecture which ties together a CDCL solver with theory solvers to form decision procedures for the SMT problem. This chapter will hopefully make clear how the theory solvers developed in later chapters fit into the larger context of an SMT solver, and the interface between the theory solver and the rest of the system.

Chapter 2 gives an introduction to the Simplex for SMT theory solver and its associated Simplex decision procedure. The chapter begins with a high level overview of the solver: its invariants, its preprocessing and its core operations. The remaining chapters of this thesis build upon this abstract introduction. After this introduction, the rest of chapter is devoted to an in-depth discussion of this algorithm and how it is implemented. This deep dive covers many topics usually considered uninteresting for research publications, but that are important for the efficiency and soundness of such solvers. Topics covered include: the delta-rational numeric representation, the soundness of inference rules built upon delta-rational arithmetic, the implementation of the tableau, the row-based propagation procedures, and how to backtrack the solver. An advantage of performing this deep dive is that it gives a sufficiently detailed view of the solver to describe our novel bookkeeping for eager conflict detection, our technique for conflict minimization, and the soundness of our model building procedure. We also go out of our away to explain the Simplex for SMT algorithm's relationship to optimization, and we show how to unify the reasoning done by this theory solver and the solver for Chapter 3 as following from a single variant of Farkas' Lemma.

Chapter 3 builds directly upon the theory solver described in Chapter 2 to describe a new sum-of-infeasibilities Simplex solver. The chapter begins with a discussion of a classical optimizing Simplex solver. We then define sum-of-infeasibilities and give an algorithm for selecting operations to minimize this function. The chapter then discusses a new heuristic conflict minimization procedure. This conflict minimization method avoids performing additional Simplex search. We give experimental results that compare the two algorithms and provide partial insights into the strengths of both. The contents of this chapter were previously published in FMCAD'13 [73].

Chapter 4 gives an introduction to theory solvers that lift the Simplex techniques given in Chapters 2 and 3 to quantifier-free linear integer arithmetic and quantifier-free linear mixed integer and real arithmetic. This includes a discussion of branching, cutting-plane generation, and rewriting within the context of SMT solving. We also give a description of more traditional branch-and-cut Mixed Integer Programming solvers for optimization. This chapter is meant as an introduction to these topics and does not include original contributions.

Chapter 5 discusses theory solvers for quantifier-free linear real, integer, and mixed integer and real arithmetic based on leveraging existing floating point linear programming and mixed integer programming solvers. We discusses an adaptation of previous work for reseeding an exact precision solver, and new methods for attempting to reproduce the implicit proofs of infeasibility coming from Mixed Integer Programming solvers. At the end of this chapter, we give an overall empirical comparison of the techniques presented in this thesis against other state-of-the-art SMT solvers. This chapter concludes with the insights we have gained from the empirical comparison, and suggested future work to extend this approach. The contents of this chapter appear in FMCAD'14 [74].

Chapter 6 concludes this thesis with an overview of the contributions in the thesis and their place in the path towards building better decision procedures for quantifier-free linear real, integer and mixed integer and real arithmetic. Additional commentary and detailed proofs are given in Appendix A.

Chapter 1

Satisfiability Modulo Theories

This chapter provides background to Satisfiability Modulo Theories (SMT) solving. Readers familiar with these topics are encouraged to skip sections as they see fit. Section 1.1 begins with a description of First-Order Logic. The SMT problem is formally defined using many-sorted first-order logic in Section 1.2. The propositional satisfiability problem is then defined as a subcase of SMT and decision procedures for this problem are described in Section 1.3. Section 1.4 gives the DPLL($\mathcal{T}$) architecture for combining SMT reasoning with efficient propositional satisfiability solvers. The final section (1.5) gives an introduction to the CVC4 SMT solver and theory solvers.

1.1 First-Order Logic

First-order logic is a language used for making precise statements. It generalizes propositional (or Boolean logic) to include functions, relations, quantifiers and variables ranging over a fixed domain of discourse. One of the great strengths of classical first-order logic is that it makes precise a notion of "truth." "Can `X` be true?" "If `X` is true, must `Y` be true?" "Is `X` always true?" First-order logic provides a foundation for making such statements and claims with mathematical precision.

This section introduces a many-sorted variant of first-order logic. When combining different domains in mathematics, one usually states what domains a variable refers to: "If `x` is an integer, then `(cons x NIL)` is a list of integers." Many-sorted logic builds upon classical [single-sorted] logic by labeling and categorizing everything in its language into explicit domains or sorts. This helps maintain clarity when dealing with multiple domains at once, and it makes many-sorted logic the natural formal basis of SMT solving.

1.1.1 Syntax

A *signature* Σ is a set of function and sort symbols, and a function ToSort that relates the function symbols to sort symbols. A *sort* or sort symbol gives a name to a domain of discourse. A signature always implicitly contains the distinguished sort symbol `Bool` for the Boolean domain. The ToSort function maps each function symbol $f \in \Sigma$ into a pair containing a k-tuple of sort symbols for the domain of f and a sort symbol indicating f's codomain. The *arity* of the function is the number k.

For example, a signature for real arithmetic is usually given as

$$\Sigma_{\mathbb{R}} = \langle \texttt{Real}, 0, 1, +, \cdot, < \rangle$$

where: `Real` is a sort symbol, 0 and 1 are 0-ary function symbols into `Real`, $+$ and $\cdot$ are 2-ary function symbols from $\langle \texttt{Real}, \texttt{Real} \rangle$ to `Real`, and $<$ is a 2-ary relation symbol from $\langle \texttt{Real}, \texttt{Real} \rangle$ to `Bool`. Function symbols with `Bool` domain like $<$ are often referred to as predicate or relational symbols (or just predicates and relations).

Included in the logic are a number of "core" or implicitly included function symbols. For every sort symbol S, we also require that there is an implicit equality symbol $=_S$ which is a 2-ary predicate symbol with the domain $\langle S, S \rangle$. The implicitly included function symbols over `Bool` are $\wedge$ (and), $\vee$ (or), and $=_{\texttt{Bool}}$ (Boolean equivalence), which are 2-ary function symbols over `Bool`, and $\neg$ (not), which is a 1-ary propositional function symbol. These symbols are known as *logical connectives*. The semantics of these symbols is given later. Note that these symbols, like `Bool`, are assumed to be in all signatures and are not explicitly written out. In the previous example, the signature $\Sigma_{\mathbb{R}}$ implicitly included the sort `Bool`, the predicate $=_{\texttt{Real}}$, the logical connective $\wedge$, etc.

First-order logic over a signature Σ includes an infinite set of variable symbols. Each variable is explicitly associated with a sort in Σ. The terms of Σ are generated by applying function symbols to terms and variables of the correct sort. We denote that the term t has sort S by $t : S$. The *terms of* Σ and their associated sorts are defined as the minimal inductive set constructed by:

- $x : S$ where x is a variable with the sort symbol S,
- the application of a function symbol $(f\ t_1\ t_2\ \ldots\ t_k) : D$ where $\text{ToSort}(f) = \langle \langle C_1, \ldots C_k \rangle, D \rangle$ and for all i, $t_i : C_i$,
- the application of a "for all" quantifier $(\forall x.t) : \texttt{Bool}$ where x is a variable symbol and $t : \texttt{Bool}$, or
- the application of an "exists" quantifier $(\exists x.t) : \texttt{Bool}$ where x is a variable symbol and $t : \texttt{Bool}$.

Throughout the thesis I will switch between LISP-like notation for function application and infix notation when appropriate and unambiguous. Additionally, as the application of a constant function symbol has no arguments, the parenthesis can be dropped without ambiguity.

The formulas of the language of Σ are the terms of sort $\texttt{Bool}$. The atoms of Σ are formulas that are either variables with sort $\texttt{Bool}$ or a function application of a non-logical connective (i.e. not the application of $\wedge$, $\vee$, $\neg$, or $=_{\texttt{Bool}}$). Literals are either an atom a or the negation of an atom $\neg a$.

Consider again the example signature $\Sigma_{\mathbb{R}}$ with variables x with sort $\texttt{Real}$ and b with sort $\texttt{Bool}$. The variable x is a term with $x : \texttt{Real}$. The symbol $1 : \texttt{Real}$ is a term constructed by the application of the constant function symbol 1 to 0 arguments. The terms $b : \texttt{Bool}$ and $(=_{\texttt{Real}}\ 0\ x) : \texttt{Bool}$ are formulas, literals and atoms while the term $(\neg\ (<\ 0\ x))$ is a formula and a literal but not an atom.

1.1.2 Semantics

The language of Σ-terms is given its semantics by *structures*. A Σ-structure contains a map from each sort symbol $S \in \Sigma$ to a non-empty set, $\mathfrak{U}(S)$ (the universe of S or its domain), and a map from each function symbol f in Σ to a function $f^{\mathfrak{U}}$ such that if $\text{ToSort}(f) = \langle\langle C_1, \ldots C_k\rangle, D\rangle$ then $f^{\mathfrak{U}}$ is a function from

$$\mathfrak{U}(C_1) \times \cdots \times \mathfrak{U}(C_k) \to \mathfrak{U}(D).$$

The built-in $\texttt{Bool}$ sort is required to have a two-element domain

$$\mathfrak{U}(\texttt{Bool}) = \left\{\mathbf{true}^{\mathfrak{U}}, \mathbf{false}^{\mathfrak{U}}\right\}.$$

The logical connectives $\wedge^{\mathfrak{U}}$, $\vee^{\mathfrak{U}}$, and $\neg^{\mathfrak{U}}$ are required to match their standard truth table definitions.

x	y	$x \wedge^{\mathfrak{U}} y$	$x \vee^{\mathfrak{U}} y$	$\neg^{\mathfrak{U}} x$
$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$
$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$
$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$
$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{true}^{\mathfrak{U}}$	$\mathbf{false}^{\mathfrak{U}}$

(1.1)

Further, the equality predicate $=^{\mathfrak{U}}_S$ for each sort symbol S is required to faithfully interpret equality over $\mathfrak{U}(S)$ i.e. the equality predicate maps $x^{\mathfrak{U}}, y^{\mathfrak{U}} \in \mathfrak{U}(S)$ to $\mathbf{true}^{\mathfrak{U}}$ iff $x^{\mathfrak{U}}$ and $y^{\mathfrak{U}}$ are the same object in $\mathfrak{U}(S)$.[1]

[1] See [49, Pages 83,127-128,140-141] for details on handling equality in Σ.

An *interpretation* M is an extension of a Σ-structure that additionally maps each variable symbol $x : S$ in the logic into $u \in \mathfrak{U}(S)$, denoted $x^M = u$. We denote by $M[x \to u]$ the interpretation that is "updated" by mapping x to u in M. More formally, $M[x \to u]$ has the same domain as M, maps every variable that is not x to the same element as M, and maps the variable $x : S$ to $u \in \mathfrak{U}(S)$. A notion of evaluation is defined for all Σ terms:

- $\mathrm{Eval}(M, x) = x^M$ for a variable x,
- $\mathrm{Eval}(M, (f\ t_1\ \ldots\ t_k)) = (f^{\mathfrak{U}}\ \mathrm{Eval}(M, t_1)\ \ldots\ \mathrm{Eval}(M, t_k))$,
- $\mathrm{Eval}(M, \forall x.\phi) = \mathbf{true}^{\mathfrak{U}}$ if for all $u \in \mathfrak{U}(S)$, $\mathrm{Eval}(M[x \to u], \phi) = \mathbf{true}^{\mathfrak{U}}$ holds (and it evaluates to $\mathbf{false}^{\mathfrak{U}}$ otherwise), and
- $\mathrm{Eval}(M, \exists x.\phi) = \mathbf{true}^{\mathfrak{U}}$ if for some $u \in \mathfrak{U}(S)$, $\mathrm{Eval}(M[x \to u], \phi) = \mathbf{true}^{\mathfrak{U}}$ holds (and it evaluates to $\mathbf{false}^{\mathfrak{U}}$ otherwise).

The $\models$ symbol is used to denote the satisfaction relation. The interpretation M satisfies the formula ϕ whenever $\mathrm{Eval}(M, \phi) = \mathbf{true}^{\mathfrak{U}}$. This is written as $M \models \phi$.

It will be clear from the context what signature Σ is currently in use. When Σ is clear from the context, instead of Σ-formula, Σ-interpretation, etc., we use "formulas", "interpretations", etc. The language of Boolean connectives is extended for convenience to include implication $\Rightarrow$, exclusive-or $\oplus$, and the constants **true** and **false**. Additionally, each sort S is required to have an *if-then-else* function symbol, ite_S, with the domain $\langle \texttt{Bool}, S, S \rangle$ and codomain S. Every interpretation M is required to satisfy the following for any $t : S$, $e : S$ and $c : \texttt{Bool}$.

$$M \models c \implies (\mathrm{ite}_S\ c\ t\ e) =_S t \quad \text{and} \quad M \models (\neg c) \implies (\mathrm{ite}_S\ c\ t\ e) =_S e$$

When the condition c of the ite_S is **true**, then the ite_S term is equal to its second child t, the "then" child. Otherwise, the ite_S term equals its third child e, the "else" child. The pattern above where s, t, c are any appropriate terms is known as a formula *schema*. The subscript $_S$ on $=_S$ and ite_S is similarly dropped when S is clear.

1.1.3 Entailment and Satisfiability

A formula ϕ *logically entails* a formula ψ if for any interpretation M such that $M \models \phi$, then $M \models \psi$. Here the symbol "$\models$" is overloaded in $\phi \models \psi$ to denote ϕ entails ψ. Formulas ϕ and ψ are *logically equivalent* whenever $\phi \models \psi$ and $\psi \models \phi$.

A formula ϕ is *satisfiable* if there exists an interpretation M that satisfies ϕ. Using the notion of logical entailment, $\phi \models \mathbf{false}$ is used to denote that ϕ is unsatisfiable. Formulas ϕ and ψ are *equisatisfiable* when ϕ is satisfiable iff ψ is satisfiable.

1.1.4 Theories

The set of *free variables* of a term t, $\text{free}(t)$, is defined inductively as:

- $\text{free}(x) = \{x\}$ if x is a variable,
- $\text{free}((f\ t_1\ t_2\ \ldots\ t_k)) = \cup_{i \in [1,k]} \text{free}(t_i)$ for function applications,
- $\text{free}(\forall x.\phi) = \text{free}(\phi) \setminus \{x\}$, and
- $\text{free}(\exists x.\phi) = \text{free}(\phi) \setminus \{x\}$.

A *sentence* is a first-order formula with no free variables. If M and M' are interpretations with the same underlying structure (i.e. they agree on everything but the variable assignment), then $M \models \phi$ iff $M' \models \phi$ for any sentence ϕ. Informally, this means that sentences are agnostic to variable assignments and only care about the structure underlying the interpretation. A formula ϕ is *valid* if it is satisfied in all interpretations. We again employ the "$\models$" symbol to denote validity by dropping the left hand side, and write $\models \phi$. Simple examples of valid formulas are propositional tautologies such as:

$$\models P \oplus Q \oplus P = Q.$$

A term t is *quantifier-free* iff t syntactically contains no quantifiers. A quantifier-free formula ϕ is equisatisfiable with its existential closure

$$(\exists x_1. (\exists x_2 \ldots (\exists x_n.\phi) \ldots))$$

where $x_1, x_2, \ldots, x_n$ is any enumeration of $\text{free}(\phi)$, the free variables in ϕ.

Theories are sets of sentences closed under logical entailment. A set of sentences S is closed under logical entailment if for all $\phi \in S$, whenever $\phi \models \psi$, then $\psi \in S$. A *model* of a theory $\mathcal{T}$ is a structure that satisfies all sentences in $\mathcal{T}$. The interpretations of a theory, $\text{Mod}(\mathcal{T})$, are all of the interpretations M that extend the models of $\mathcal{T}$.[2]

1.2 The SMT Problem

The *Satisfiability Modulo Theories* problem is to decide whether a formula ϕ has a satisfying interpretation in the theory $\mathcal{T}$. A formula ϕ is satisfiable modulo $\mathcal{T}$ if there is an interpretation $M \in \text{Mod}(\mathcal{T})$ such that $M \models \phi$. We write $M \models_{\mathcal{T}} \phi$ as shorthand for $M \in \text{Mod}(\mathcal{T})$ and $M \models \phi$.

[2] Note that most authors use $\text{Mod}(\mathcal{T})$ to reflect the models of $\mathcal{T}$ instead of the interpretations. Hence the notation $\text{Mod}(\cdot)$.

Theories of interest are ones that are expressive enough to model interesting problems, but also have efficient decision procedures. What distinguishes SMT solvers from other automated first-order logic solvers is that SMT solvers build efficient specialized solvers for theories of particular interest instead of focusing on more general methods (e.g. first-order resolution). In order to develop specialized solvers, the theory $\mathcal{T}$ [or theories] the solver can solve are built into the solver (as opposed to being specified by input axioms). For example, the theory of arrays allows for reasoning about memory reads and writes, the theory of fixed-width bitvectors naturally encodes common CPU arithmetic instructions, and the combination of these two theories gives a natural logic for encoding assembly instructions. Developing new theories allows for new domains to be expressed succinctly by developing specialized procedures. The core set of theories that most state-of-the-art general-purpose SMT solvers support are:

- Booleans,
- uninterpreted functions,
- linear mixed real and integer arithmetic,
- fixed-width bitvectors,
- arrays, and
- inductive datatypes.

The parametric theories, such as uninterpreted functions, arrays, and datatypes, can be used to mix together base theories, such as bitvectors and arithmetic, into combined theories. (See Sec. 1.5.5 for more on combination.)

Theories explicitly restrict the semantics of function symbols given by the interpretations. Much in the same way that the function symbol $\wedge$ is restricted so that $\wedge^{\mathfrak{U}}$ is indistinguishable from the standard "and function," theories restrict the semantics of their *interpreted* symbols such as $+$, 0, $<$, etc. These interpreted symbols are restricted by the set of sentences in the theory referencing the symbols (called axioms). Examples of axioms are statements such as 0 is the additive identity of $+$:

$$\forall(a : \texttt{Real}) a + 0 = a$$

or that addition preserves the order of $<$

$$\forall(a, b, c : \texttt{Real}) a < b \implies a + c < b + c.$$

These axioms require that the functions $+$ and 0 have certain properties that make them behave according to our expectations on $+$ and 0. Fixed-width bitvectors on the other hand are finite and can be defined using a *standard* model where the bitvector operations are defined by a propositional equivalent circuit. However, fully defining theories in either of these fashions is outside of the scope of this thesis. (Interested readers are directed to [65, Section 2].) When designing an SMT solver, it is of great benefit to design the solver with respect to a standard model. This fixes how to implement the operations and greatly simplifies reasoning. The standard models for many theories, however, are infinite. The intention of the axioms of the theory is to restrict the function symbols to act like their counterparts in the standard interpretation. There may be some concern that a formula could be satisfiable in the theory, but not hold in the standard model. Fortunately, the theories the later chapters cover, such as real arithmetic, are all *complete*. A complete theory is one in which every sentence is logically equivalent modulo the interpretations of the theory. (For every pair of models A, B of the theory and any Σ-sentence ϕ, $A \models \phi$ iff $B \models \phi$.) This means that models of the theory cannot be distinguished by Σ-sentences. In arithmetic, this means that the function $+^{\mathfrak{U}}$ cannot be distinguished over $\mathfrak{U}$ from the standard $+$ function over the $\mathbb{R}$ by a $\Sigma_{\mathbb{R}}$ formula. If a theory is complete with a standard model, this significantly simplifies the design of a theory solver. The solver may then be designed around the standard model as a formula is satisfiable iff it is satisfiable in the standard model. This allows for both the theory solver designer [and the user] to treat such interpreted theory symbols "+" as if they are the mathematical function. (Some theories of interest are not complete. See Chapter 4.)

Signatures Σ may be explicitly extended with new *uninterpreted function* symbols. For example, to declare an uninterpreted function `f` from an integer and a Boolean to bitvectors with 2 bits in the SMT-LIBv2.0 language, we write

```
(declare-fun f (Int Bool) (_ BitVec 2))        (1.2)
```

These are not interpreted by the theory. All that is known about these functions is that they act like functions over their domains. Given the declaration (1.2), `(f 5` **true**`)` is well-formed while `(f 5 0)` is not.

Many of the notions from first-order-logic can be extended modulo $\mathcal{T}$ using the restricted notion of satisfaction modulo $\mathcal{T}$:

- entailment modulo $\mathcal{T}$ ($\phi \models_{\mathcal{T}} \psi$): for every $M \in \text{Mod}(\mathcal{T})$, if $M \models_{\mathcal{T}} \phi$, then $M \models_{\mathcal{T}} \psi$,
- validity modulo $\mathcal{T}$ ($\models_{\mathcal{T}} \phi$): if $M \in \text{Mod}(\mathcal{T})$, $M \models_{\mathcal{T}} \phi$, and
- equisatisfiability modulo $\mathcal{T}$, ϕ is satisfied by some $M \in \text{Mod}(\mathcal{T})$ iff ψ is satisfied by some $M' \in \text{Mod}(\mathcal{T})$.

Past this point, the modifier "modulo $\mathcal{T}$" may be dropped for brevity when it is clear from the context that we are working modulo the theory $\mathcal{T}$.

A *decision procedure* is a terminating algorithm that when given an input produces either a yes or a no answer to a formal problem. A *semi-decision procedure* always reports yes when the answer is yes, but may fail to report no when the answer is no. An *SMT solver* is a procedure for deciding whether ϕ is satisfiable (**Sat**) or unsatisfiable (**Unsat**) for a fixed theory $\mathcal{T}$. In general, an SMT solver may not even be a semi-decision procedure and may in addition to answering **Sat** or **Unsat** may report unknown or fail to terminate. An SMT solver is *sound* if whenever it reports **Sat**, then the input ϕ is satisfiable, and whenever it reports **Unsat** the input ϕ is unsatisfiable. An SMT solver is *complete* if it always either reports **Sat** or **Unsat**. A theory $\mathcal{T}$ is *decidable* if has a sound and complete procedure for checking satisfiability. Many SMT theories of interest are in general undecidable, i.e. they cannot have a decision procedure. (See Chapter 4.)

SMT solvers implement effective-in-practice procedures to handle fragments of their input languages. These fragments are known as *logics*. Most SMT solvers focus on building satisfiability procedures for quantifier-free formulas. Quantifier reasoning is then built on top of the decision procedures for the quantifier-free fragments by performing sound but incomplete instantiation [43, 96]. Instantiation may be made complete in many important applications [102]; however, this requires proving locality properties for fixed quantifiers. A theory admits *quantifier elimination* iff for all formulas ϕ there exists a logically equivalent quantifier-free formula ϕ_{QF}. If there exists a procedure for computing ϕ_{QF} given ϕ and the quantifier-free satisfiability problem is decidable, then quantifier elimination forms a decision procedure for the theory. All theories that admit quantifier elimination are complete.[3]

Logics can additionally limit what kinds of terms can be constructed. For example, linear arithmetic can be defined by not allowing any multiplication symbols, and difference logic further restricts arithmetic to only have atoms of the from $x - y \leqslant c$ where c is a rational constant.

Instead of just deciding the satisfiability of ϕ and answering **Sat** or **Unsat**, SMT solvers may also be *model producing* and *proof producing*. A model producing SMT solver, in addition to answering **Sat**, returns along with the answer a handle H to a satisfying interpretation M, (**Sat** H). The handle H is a function that maps any term t into t^M. In simple cases, H is just an assignment of each variable x that appears in ϕ. For example,

$$\phi : x > 0 \wedge y \leqslant 5 \wedge ((x = y + 5) \vee (x = y - 5))$$

is satisfied by the handle $[x \to 1]$, $[y \to -4]$. The additional level of indirection

[3] See [64] for more on quantifier elimination.

produced by using handles instead of variable assignments, allows for supporting infinite representations and the additional computation sometimes needed to refine abstractions of interpretations into interpretations.

1.3 Decision Procedures for SAT

The SMT problem is a generalization of the propositional satisfiability (SAT) problem. The SAT problem may be viewed as a special case: the satisfiability of ϕ where the only sort in ϕ is `Bool` and ϕ is quantifier-free. The SAT problem is the classic **NP**-complete problem.

A *clause* is a finite disjunction of literals. For example, $(p \vee q \vee r)$ is a clause over the propositional variables p, q and r. Viewing clauses as sets instead of just binary disjunctions simplifies quite a bit of reasoning around associativity and commutativity of $\vee$. For simplicity, we additionally allow clauses with 0 or 1 literal. A clause with 0 literals is called the empty clause and is equivalent to **false**. Clauses with 1 literal are called *unit clauses*. A formula ϕ is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. The following is a CNF formula over the propositional variables p, q and r:

$$(p \vee q \vee r) \wedge (\neg p) \wedge (\neg q). \tag{1.3}$$

Currently, the best performing general decision procedures for the SAT problem work by first reducing a SAT formula ϕ to an equisatisfiable CNF formula ϕ', and then using a decision procedure for CNF formulas. The reduction to CNF may be implemented by a straightforward recursive algorithm that works by adding for every subformula ψ an additional variable x_ψ such that in all satisfying models M, $M \models \psi = x_\psi$, and encoding each equivalence in CNF [109]. The result of applying this reduction to a propositional formula ϕ is an equisatisfiable formula ϕ'. A major advantage of working in CNF is that it provides a uniform representation that enables specialized techniques and simpler reasoning compared to an arbitrary structure. If any of the literals in the clause is known to be **true**, the clause is satisfied and the rest of the clause can be ignored. Equivalently, all of the literals in the clause must be **false** for the clause to be unsatisfied. Section 1.5.11 gives rewriting rules that simplify a formula in CNF to either not contain either the constants **true** or **false**, or to be exactly the constants **true** or **false**. The CNF solver can then be assumed to only have to deal with CNF formulas not containing constants.

The simple but powerful *resolution rule* takes two clauses $C' \equiv \{x\} \cup C$ and

$D' \equiv \{\neg x\} \cup D$ and infers a new clause $C \cup D$ that does not contain x:

$$\frac{\{x\} \cup C \qquad \{\neg x\} \cup D}{C \cup D}\ \textsc{Resolution} \tag{1.4}$$

Inference rules like this are read as follows: whenever the top (the *antecedents*) holds then the bottom (the *consequent*) holds as well. If M satisfies C' and D', then M must satisfy $C \cup D$ regardless of whether x evaluates to **true** or **false** in M. Sound inferences only derive valid entailments: if ϕ infers ψ, then $\phi \models \psi$. All of the inference rules given will be sound, but most will not be proven.

The Davis-Putnam (DP) procedure is a sound and complete procedure for SAT that uses resolution as its primary workhorse [35, 37]. The procedure is given as input a set of clauses $\mathtt{C_{DB}}$. DP selects a `Bool` variable x that appears in some clause, and then exhaustively applies resolution over the clauses in $\mathtt{C_{DB}}$ over the variable x to infer a new set of clauses that do not contain x. The union of these new clauses with the clauses that did not previously contain x are equisatisfiable with the original set of clauses. If the procedure ever derives the empty clause, then the problem is **Unsat**. Resolution can only get stuck if a variable is *pure*, all instances of a variable x in the clauses are either *positive* (x) or *negative* ($\neg x$). Such pure variables can be safely eliminated by effectively assigning them to either **true** or **false** respectively. The clauses containing a pure variable x can be dropped or ignored as the result is an equisatisfiable set of clauses. DP then recursively checks the resulting set of clauses $\mathtt{C_{DB}}'$ (from either resolution or pure variable elimination) for satisfiability. If all variables have been eliminated, then $\mathtt{C_{DB}}$ is either empty in which case the problem is **Sat**, or $\mathtt{C_{DB}}$ contains the empty clause and is **Unsat**.

The Davis-Putnam-Logemann-Loveland (DPLL) SAT procedure takes the DP procedure and reorients its focus to recursively look for models [35, 36]. (Pseudocode for DPLL is given in Fig. 1.1.) DPLL tries to find a satisfying partial model to a set of clauses, $\mathtt{C_{DB}}$, over n Boolean variables. It does this by recursively guessing an assignment to each variable and checking whether the set of guesses made satisfy the clauses. To improve upon naively enumerating and testing all 2^n variable assignments, DPLL tries to do better by using one guess to learn the assignments of more variables before guessing again. An assignment of a variable x to constant value v will be denoted $[x \to v]$ (following the notation for substitution). The guess or *decision* that the Boolean variable x gets the value $v \in \{\textbf{true}, \textbf{false}\}$ is a marked variable assignment $[x \to v]_d$. The decision $[x \to v]_d$ is added to a stack of partial assignments called a `trail`. The *decision level* is the total number of decisions on the trail. Appending a variable assignment $[x \to v]$ to the trail simulates adding either the unit clauses $\{x\}$ (for $v = \textbf{true}$) or $\{\neg x\}$ for ($v = \textbf{false}$). After adding the decision, the procedure performs a restricted form of resolution called *unit propagation* that only derives unit clauses.

$$\textsc{Unit-Prop}\quad \frac{p_1, \ldots, p_n \qquad \neg q_1, \ldots, \neg q_m \qquad \neg p_1 \vee \ldots \vee \neg p_n \vee q_1 \vee \ldots \vee q_m \vee r}{r} \tag{1.5}$$

A derived unit clause r effectively forces the variable x appearing in the clause to be assigned to a value v. If r is positive (a variable x), then x must be assigned to **true**, while if r is negative ($\neg x$), then $v =$ **false**. Whenever a new unit r and $[x \to v]$ is learned, then either:

- x is not assigned in `trail` and $[x \to v]$ is added to `trail` as a propagation.
- If $[x \to v]$ is already on `trail`, r is satisfied and can be ignored,
- Or if $[x \to \neg v]$ is already on `trail`, then r cannot be satisfied, and this branch must be **Unsat**.

Once a branch is known to be **Unsat**, the procedure terminates and returns **Unsat**. Unit propagation is applied to a fix-point – either the rule has derived a contradiction or the rule cannot derive any new unit clauses. If it is not the case that all variables appear in the trail, another variable is selected for branching and the process recursively continues. If all variables are in the trail, Unit-Prop is exhausted, and there are no conflicts, the branch [and the original problem] is **Sat**.

Efficient implementations of DPLL can substantially outperform naive implementations. No new copies of the input clauses need to be made or removed as nothing but unit clauses are derived. Those can directly be handled with the trail. The program's recursion may be fully simulated by using the trail and marking decision literals differently than propagations. Recursion amounts to pushing a new decision $[x \to v]_d$ onto the trail and restarting the main loop. To simulate returning, the solver backtracks to the previous decision, flips the assignment of that decision, removes the decision marking ($[x \to \neg v]$), and continues as normal. If there is no decision to be flipped on the trail, all of DPLL's branches have been explored and the problem is **Unsat**. More significantly there are efficient datastructures such as watched literals and clause databases that can make Unit-Prop very efficient. Further lines of research examine heuristics for which variable to choose as a decision and what value it should be assigned, and when to restart the SAT search [14, 66, 86, 94]. This non-recursive, non-destructive and efficient reformulation eliminates a large amount of overhead.

The Conflict Driven Clause Learning (CDCL) framework provides a modern, sound and complete procedure for SAT that combines the strengths of DP

```
DPLL($C_{DB}$, trail):
  apply UNIT-PROP using trail and $C_{DB}$ to a fix-point
  if both $[x \to \textbf{true}]$ and $[x \to \textbf{false}]$ for any x:
    return Unsat
  if AllClausesAreSat($C_{DB}$, trail):
    return Sat
  else:
    Select some x such that x is not on trail
    $\texttt{trail}_{\textbf{true}} \leftarrow \texttt{trail} \cup [x \to \textbf{true}]_d$
    if DPLL($C_{DB}$, $\texttt{trail}_{\textbf{true}}$) == Sat:
      return Sat
    $\texttt{trail}_{\textbf{false}} \leftarrow \texttt{trail} \cup [x \to \textbf{false}]_d$
    if DPLL($C_{DB}$, $\texttt{trail}_{\textbf{false}}$) == Sat:
      return Sat
    return Unsat
```

Figure 1.1: Psuedocode for the DPLL algorithm.

and DPLL [81,101]. CDCL uses a DPLL-style search for satisfying assignments by making decisions and applying Unit-Prop. CDCL additionally associates with each propagated literal the id of the clause that propagated it as part of the trail, $[x \to v]_{id}$. What distinguishes CDCL from DPLL, is that whenever CDCL derives both $[x \to \textbf{true}]$ and $[x \to \textbf{false}]$, it learns a clause that heuristically explains this inconsistency or conflict. CDCL specifically tries to learn a *conflict clause*$C = \{\neg\ell_1, \neg\ell_2, \dots, \neg\ell_k\}$ where each ℓ_i is currently true in the trail, and $C_{DB} \models C$ (as $C_{DB} \wedge \neg C \models \textbf{false}$). A simple algorithm for guessing C is the negation of all of the decisions. The solver can safely add such a C to the database as

$$C_{DB} \models C \text{ implies that } C_{DB} \text{ is logically equivalent to } C_{DB} \wedge C.$$

The solver then backtracks to the lowest decision level such that C would have been propagated by UNIT-PROP if it had been in C_{DB} before. (More formally, the prefix of `trail` up to the decision level ℓ can apply UNIT-PROP to C to assign a variable x to v and previously at level $\ell' > \ell$, x was assigned $\neg v$. The variable x is known as the Unique Implication Point [101].) The search then proceeds again from this point. The conflict now propagates at least one literal in opposite polarity, cutting off the previous set of decisions. This naive conflict generation algorithm almost simulates DPLL. More advanced algorithms for generating conflict clauses examine the derivation of both $[x \to \textbf{true}]$ and $[x \to \textbf{false}]$. The trail and the clause ids associated with each Unit-Prop propagation define an implicit implication graph. Conflict analysis algorithms examine this implication graph in order to generate conflicts [81,101]. The result is smaller and less redundant conflicts that cut off significantly more future search space. These algorithms can be viewed as performing highly selective forms of resolution to derive C. CDCL solvers alternate between this search for satisfying assignments, and learning from its failures to derive new facts to guide future search.

1.4 DPLL($\mathcal{T}$)

The most common architecture for building general purpose SMT solvers is DPLL($\mathcal{T}$) [11,54,90,99]. The architecture weds together a DPLL or CDCL-style SAT solver augmented with $\mathcal{T}$-specific reasoning to produce an SMT decision procedure. Part of the reason for the popularity of DPLL($\mathcal{T}$) solvers is that the architecture cleanly separates propositional reasoning from theory reasoning. The theory-specific reasoning is handled by a *theory solver*. Theory solvers are modules that can perform certain classes of reasoning over $\mathcal{T}$. The core facility a theory solver provides is to decide the satisfiability of conjunctions of $\mathcal{T}$-literals.

An augmented SAT solver is used to solve the Boolean abstraction of the formula where every theory atom [and quantified formula] is handled as a propositional variable. These are *asserted* to the theory solver as literals. The theory solver then checks the $\mathcal{T}$-satisfiability of the set of asserted theory literals, $\mathcal{A} = \{p_1, \dots, p_M\}$. If the asserted literals are satisfiable and the SAT solver has found a propositionally consistent assignment to the Boolean abstraction, then the original input formula is satisfiable. However, if $\mathcal{A}$ is inconsistent, then some subset C of $\mathcal{A}$ is inconsistent and the negation of C is $\mathcal{T}$-valid.

$$\frac{\bigwedge_{p_i \in C} p_i \models_{\mathcal{T}} \mathbf{false}}{\models_{\mathcal{T}} \bigvee_{p_i \in C} \neg p_i} \textsc{Theory-Conflict} \tag{1.6}$$

The derived clause $\bigvee_{p_i \in C} \neg p_i$ is $\mathcal{T}$-valid and is sent to the SAT solver. The SAT solver then adds this clause to its clause database. This newly-added clause is (by construction) falsified in the SAT solver, and a modified version of conflict clause generation is invoked to backtrack the SAT solver. If the SAT solver ever concludes that its clause database is infeasible, then the valid theory lemmas and the original propositional structure are enough to infer that the original input is **Unsat**.

There are many candidate modes of operation and interplay between SAT solvers and theory solvers. One popular explanation of the relationship between the SAT solver and the theory solver is that the theory solver contains an infinite set of theory-valid *lemmas* that the SAT solver is querying on demand [11,99]. Three naive modes of operation that are easy to explain are fully *lazy*, fully *eager*, and *in-the-loop*. A fully eager strategy enumerates all possibly relevant theory lemmas before SAT solving. Instances of fully eager strategies in practice include Ackermanization for uninterpreted functions, and some bit-blasting strategies for fixed-width bitvectors [11, Section 26.3]. Ackermanization adds the lemmas that enforce that the result of two instances of a function are equal whenever all of the arguments are equal:

$$\left(\bigwedge_{i=1}^{k} t_i = s_i\right) \implies (f\ t_1\ \ldots\ t_k) = (f\ s_1\ \ldots\ s_k).$$

Fully lazy on the other hand waits for the SAT solver to build a full Boolean satisfying assignment before checking for theory consistency. In-the-loop solving checks for theory satisfiability at specific points throughout the SAT search: before making a decision and before concluding SAT. Most general-purpose SMT solvers opt for carefully tuned balances of spreading inferences over eager, lazy or in-the-loop flavors of reasoning where it has been deemed experimentally appropriate.

This section has so far only described a minimal abstract DPLL($\mathcal{T}$) interface calculus where the theory solver provides only satisfiability checks. The next section will implicitly describe the full abstract DPLL($\mathcal{T}$) calculus via the SMT solver CVC4.

1.5 The CVC4 Architecture

CVC4 is a state-of-the-art Satisfiability Modulo Theories solver [7]. CVC4 is the fourth in the line of the Cooperating Validity Checker (CVC) SMT solvers which supplanted the Stanford Validity Checker [6,9,13,103]. The implementation of CVC4 is a refinement of Abstract DPLL($\mathcal{T}$) [90]. This section gives a high level description of the architecture of CVC4.

1.5.1 User Interaction

CVC4 provides users with a native C++ binary interface. Built on top of this interface are several APIs for other programming languages and textual interfaces for the languages: SMT-LIB version 1.2 [95], SMT-LIB version 2.0 [12], TPTP [105] and the CVC presentation language. The most popular language for describing SMT problems is the SMT-LIB 2.0 language. The core of the user interface can be defined using a key subset of the SMT2 commands. The user interacts with CVC4 though the following major commands:

- (**set–logic** *LogicString*):
  This command specifies what logic (Sec. 1.2) the problem is over. This informs the solver of what theories and combinations of theories may be used (additionally it may change what decision procedure is used). If this is left unspecified, CVC4 assumes an "ALL_SUPPORTED" logic that enables the widest range of supported formulas. This command must precede all of the other commands in this list.

- (**push**):
  This increases an integer called the *user-context-level*. Each context-level forms a scope for declarations and assertions. The user-context-level is initially zero.

- (**pop**):
  This decreases the user-context-level. If the context-level ever becomes negative, the input is malformed.

- (**declare–fun** *Name* ([`Sorts`]) `Sort`):
  This declares an uninterpreted function symbol "*Name*" that operates over a list of sorts that form the domain of the function and the sort of the codomain of the function. Variables are declared as uninterpreted constants by leaving the domain list empty, e.g. (**declare–fun** `x` () `Real`).

- (**assert** ϕ):
  Asserts that the formula ϕ holds in the current user assertion level. This is implicitly pushed onto the back of a stack of user assertions, $\mathcal{A}_{\mathrm{U}}$. This stack will be automatically snapped back to the correct length upon a user (**pop**).

- (**check–sat**):
  The solver checks for the $\mathcal{T}$-satisfiability of $\mathcal{A}_{\mathrm{U}}$ and returns either: **Unsat**, **Sat** or **Unknown**.

- (**get–model**):
  After a (**check–sat**) call that answers **Sat**, a model-producing SMT solver (like CVC4) can return a handle H to a satisfying interpretation.

Figure 1.2 gives an abstraction of the C++ public interface that CVC4 provides for these calls. Most of these actions are reasonably straightforward with almost all of the work being delegated to SMTCHECKSAT. There are many user queries, like (**get–model**), the solver can answer after a SMTCHECKSAT to help the user

```
class SmtEngine {
  void setLogic(string logic);
  void push();
  void pop();
  Result assertFormula(Expr e);
  Result smtCheckSat();
  Model getModel();
};
```

Figure 1.2: Abstraction of the CVC4 SmtEngine.

in a variety of tasks such as proofs, unsat cores, assignments, and lemma-lifting [29]. These are outside of the scope of this thesis.

1.5.2 SMTCHECKSAT

The SMTCHECKSAT routine is responsible for orchestrating the overall solving process. It is responsible for simplifying the input problem, setting up the problem to be solved, dispatching the problem to the decision procedures and saving the results of the decision procedures for future user queries.

The SMTCHECKSAT routine is given in Fig. 1.3. The routine begins by making a copy of the user assertions $\mathcal{A}_{\mathrm{U}}$ as a list of internal equisatisfiable assertions $\mathcal{A}$. Working on a copy frees $\mathcal{A}$ to be rewritten into more efficient forms without necessarily maintaining logical equivalence across user (**pop**) operations. The handle to the model H is maintained to translate between interpretations for $\mathcal{A}$ and the user input. The internal assertions are then *preprocessed*. Preprocessing transforms $\mathcal{A}$ with the goal of making the problem more efficiently solvable. The assertions are then further transformed by *theory sanitization*. The full language that SMT solvers support is not supported directly by the theory solver. Sanitizing reduces these more complex expressions that theory solvers do not support to [equisatisfiable] terms that theories support. (Preprocessing and sanitizing are discussed more in Sections 1.5.8 and 1.5.9.) SMTCHECKSAT then sends all of the sanitized assertions to the PROPENGINE. The PROPENGINE is an abstraction of the SAT solver in DPLL($\mathcal{T}$) while the THEORYENGINE abstracts the theory $\mathcal{T}$ in DPLL($\mathcal{T}$). The PROPENGINE works with the THEORYENGINE (see Sec. 1.5.5) to then decide satisfiability. (See sections 1.5.3 and 1.5.5.) Based on the result of this call, SMTCHECKSAT, performs some post-processing, caches the post-processing, and returns the result.

```
Result SmtEngine::checkSat() throw(...){
  $\mathcal{A}$ := copy $\mathcal{A}_U$;
  Model H := empty
  preprocess($\mathcal{A}$, H);
  for each p in $\mathcal{A}$:
    theoryPurify(p, H);
  for each p in $\mathcal{A}$:
    propEngine.assert(p, H)
  res := propEngine.checkSat()
  if res is Sat:
    theoryEngine.finalizeModel(H, M)
    store H
  ... // Save info for other user queries
  return res
}
```

Figure 1.3: Abstraction of the CVC4 SMTCHECKSAT routine.

1.5.3 PROPENGINE

The PROPENGINE abstracts deciding propositional satisfiability of the assertions $\mathcal{A}$ sent to it. The PROPENGINE works together with the THEORYENGINE to then decide the satisfiability of $\mathcal{A}$. Figure 1.4 visualizes the flow of communication between the PROPENGINE, the THEORYENGINE, and the individual theory solvers. The assertions in $\mathcal{A}$ are efficiently converted into equisatisfiable clauses using a caching CNF converter ToCnf. The CNF converter allocates new propositional variables for inner Boolean terms when necessary. The cache of the converter creates a one-to-one correspondence between theory atoms and a propositional variable. A version of the main solving loop of CDCL which has been extended for theory reasoning is given in Sec. 1.5.4. Whenever the propositional variable correspondence to a theory atom is set to a value, the THEORYENGINE is sent the assertion as a literal. The THEORYENGINE then communicates back the theory satisfiability of these assertions using *theory lemmas*. A theory lemma is a theory-valid formula ϕ that the theory solver is requiring the PROPENGINE to satisfy in the propositional abstraction. The PROPENGINE adds the lemma ϕ to its own assertions, converts it to CNF, and integrates these clauses into the state of the CDCL search. (Lemmas are sanitized before going to the PROPENGINE.) Theory lemmas include theory conflicts, splitting-on-demand lemmas, quantifier instantiations, and other forms of theory lemmas in

CVC4.[4] When the theory solvers do not emit lemmas, this can either be interpreted as **Sat** or **Unknown** depending on the context.[5]

Splitting-on-demand is a technique that allows theories to request that the SAT solver locally enumerate cases [10]. For example, if the literal $x \neq y$ is sent to the theory of real arithmetic, but under the current candidate handle H, x and y are assigned the same value, the solver can request the split:

$$x = y \vee x < y \vee x > y$$

The SAT solver then decides that either $x > y$ or $x < y$ is true and sends this to the theory solver. This simplifies solver design as the solvers do not need to build the additional machinery to enumerate these cases, to recombine the individual proofs or to design a decision procedure that natively handles the entire input language.

The THEORYENGINE also *propagates* theory literals to the PROPENGINE that have been inferred using the current assertions.[6] During the conflict analysis phase, PROPENGINE can ask the THEORYENGINE for an *explanation* of the propagated literals as a conjunction of previous assertions.

$$\text{explanation}(l) = \bigwedge_{a_i \in \mathcal{A}} a_i \text{ such that } \bigwedge a_i \models_{\mathcal{T}} l$$

This allows the CDCL solver to use the clause explanation$(l) \implies l$ to explain the propagation of l lazily.

1.5.4 CDCL($\mathcal{T}$)

The CDCL loop of CVC4 requires tightly interweaving theory reasoning at crucial steps. The psuedocode for combining CDCL with theories is given in Fig. 1.5. The current version of CVC4 1.4 uses minisat 2.2.0 with adaptations for use with DPLL($\mathcal{T}$) as its underlying SAT solver. The two crucial abstract datastructures are the clause database, C_{DB}, and the trail of SAT propagations, assignments and theory propagations, `trail`. The procedure unfolds roughly the same way as CDCL (Sec. 1.3). The main loop begins by saturating UNIT-PROP, and if inconsistent, learning a conflict and then backtracking (lines 4-11). Running UNIT-PROP to a fix-point is often referred to as Boolean Constraint Propagation (BCP). As the SAT variables are assigned in `trail`, the variables corresponding to theory atoms and their assignments are sent to THEORYENGINE as

[4] Andrew Reynolds' thesis discusses how quantifiers are handled in CVC4 [96].

[5] See Section 1.5.4 for a discussion of when this means **Sat** or **Unknown**.

[6] CVC4 currently follows DPLL($\mathcal{T}$) and requires the propagated literals to have corresponding SAT variables.

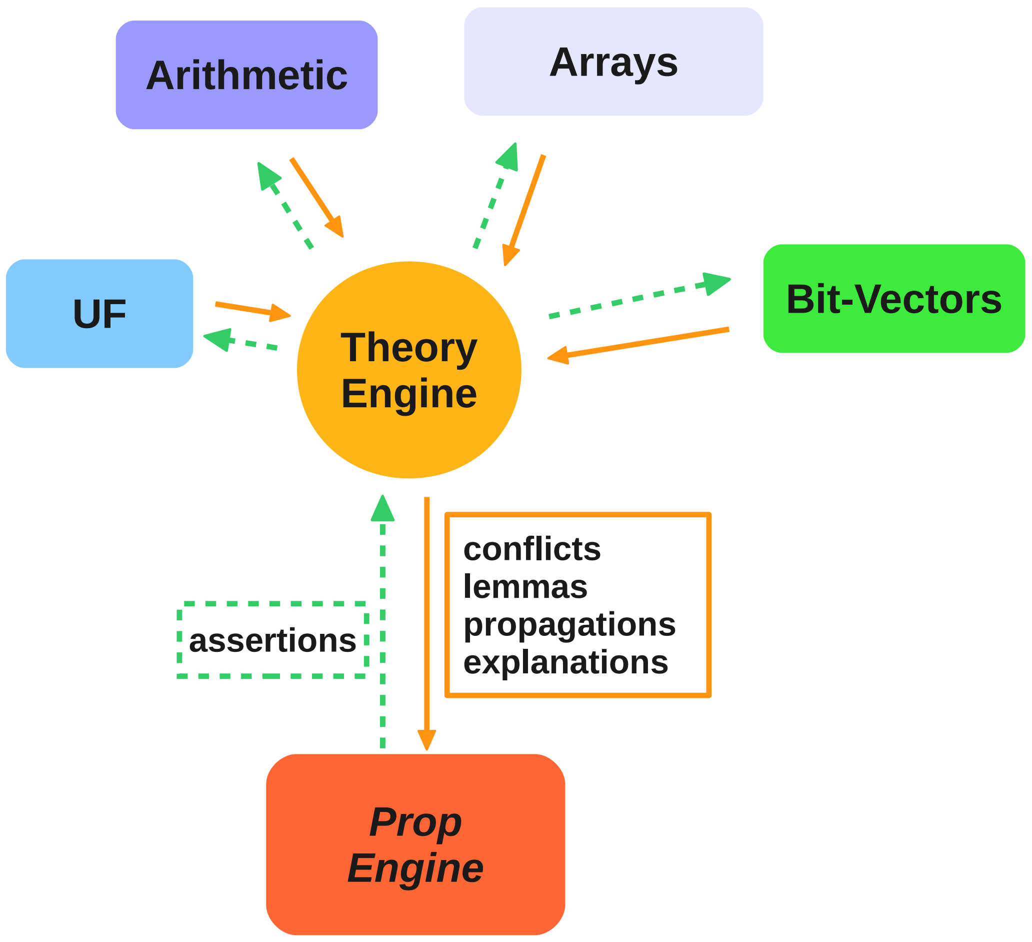


Figure 1.4: A visualization of the DPLL($\mathcal{T}$) communication in CVC4. (Courtesy Liana Hadarean)

assertion literals and pushed onto a stack of assertions $\mathcal{A}$. If the partial assignment `trail` is consistent with $\mathtt{C_{DB}}$, the TheoryEngine is called for an in-the-loop consistency check, known in CVC4 as a *standard effort* check (line 12). A standard effort check may report either:

1. $($**Sat** $\mathrm{H})$ if $\mathcal{A}$ is $\mathcal{T}$-satisfiable,
2. $($**Unsat** $\mathrm{C})$ if $\mathcal{A}$ contains a $\mathcal{T}$-conflicting subset C,[7]
3. $($**Lemma** $\phi)$ where ϕ is a $\mathcal{T}$-lemma, or
4. **Unknown** if the theory solver decides the check is too expensive.

What is done in standard effort checks is heuristic. The solver may always return **Unknown**. What is appropriate depends on the theory solver. The standard effort checks are not required for correctness, but are useful for reducing the search space explored earlier than if the checks are only performed at leaves. If there are any pending theory lemmas, these are imported into $\mathtt{C_{DB}}$ and the main loop is restarted to check for conflicts and further propagation (lines 13-15). If there are no pending lemmas, TheoryEngine is asked to propagate theory literals (line 16). If theory literals were propagated, the main loop is restarted to again check propositional consistency of the clauses and to propagate more (lines 17-18). If the clause database $\mathtt{C_{DB}}$ is satisfied by the partial assignment `trail`, the propositional abstraction has been satisfied. The TheoryEngine then does a *full effort* check. A full effort check is required to report either

1. $($**Sat** $\mathrm{H})$ if $\mathcal{A}$ is $\mathcal{T}$-satisfiable,
2. $($**Unsat** $\mathrm{C})$ if $\mathcal{A}$ contains a $\mathcal{T}$-conflicting subset C, or
3. $($**Lemma** $\phi)$ where ϕ is a $\mathcal{T}$-lemma and (*) ϕ is not currently propositionally satisfied.[8]

Full effort checks are required for the correctness of reporting satisfiable. The TheoryEngine and theory solvers must commit to either reporting the set of assertions is satisfiable, unsatisfiable, or if theory solver is unsure, it must send out a lemma that forces the SAT solver to make additional decisions (hence the new (*) condition).

If $($**Sat** $\mathrm{H})$ is reported for the theory, the CDCL solver returns **Sat** with the current trail `trail` as the propositional variable assignment and the CDCL($\mathcal{T}$)

[7] The response $($**Unsat** $\mathrm{C})$ is implemented as a wrapper around $\left(\textbf{Lemma } \bigvee_{\ell \in C} \neg \ell\right)$ with additional debugging and logging.

[8] Formally, the condition (*) on full effort lemmas is that the CNF abstraction of ϕ, $\mathrm{ToCnf}(\phi)$, is not satisfied by the current trail, `trail`.

solver can combine H and `trail` to build a satisfying interpretation for the user input assertions. If there are lemmas of the form (**Unsat** C) or (**Lemma** ϕ), the main loop is again restarted. Finally, if the C_{DB} is not yet satisfied by the `trail`, the solver pushes the sat-level and decides a value for an unassigned variable. Note that removing lines 12-20 and 20-24 results in the standard abstraction of CDCL.

1.5.5 THEORYENGINE

General purpose SMT solvers such as CVC4 support multiple input theories $\mathcal{T}$. The THEORYENGINE provides an abstraction of the solvers for many different theories as a single combined theory. It is responsible for being an intermediary between the PROPENGINE and the theory solvers, dispatching atoms to the correct theory solver, and performing theory combination. Literals such as $s < t$ are dispatched to the theory solver for arithmetic while $(f\ x) = (f\ (f\ x))$ is sent to uninterpreted functions. This maintains clear lines of separation for implementing different theories. Suppose that f's codomain is `Real`, there could be atoms where it is unclear which theory to dispatch to such as $(f\ x) < t$. This makes the term $(f\ x)$ *shared* between two theories that must agree on what $(f\ x)$ is if the problem is satisfiable. The THEORYENGINE acts as the *combination* of the participating theories. CVC4 uses a modern version [68, 108] of the classical Nelson-Oppen method [87]. Under some restrictions on the theories, Nelson-Oppen ensures that if both theories agree on an alignment, either $(s = t)$ or $(s \neq t)$ for all shared terms s and t (of the same sort), then the procedure can generate a combined model and soundly conclude **Sat** for the combination. CVC4 further optimizes this method using lazy combination [20] and care graphs [68] to reduce the number of equalities guessed.

1.5.6 Theory Solvers

Theory solvers are procedures for deciding the satisfiability of conjunctions of assertions $\mathcal{A}$ in a logic for a theory $\mathcal{T}$. An abstracted interface for a theory solver is given in Fig. 1.6. Theory solvers receive new assertions from the THEORYENGINE. When the PROPENGINE wants to know the $\mathcal{T}$-consistency of its assertions, it requests the THEORYENGINE to perform a `TheorySolver::check(·)` which the THEORYENGINE sends to each theory solver. Both standard effort and full effort checks output lemmas onto an *output stream* that the THEORYENGINE polls and forwards to the PROPENGINE after each check. As discussed in Section 1.5.4, standard effort checks are in-the-loop and not required for consistency. Full effort checks, on the other hand, are required to output a conflict

```
CDCL($\mathcal{T}$):
  sat-level := user-level
  loop:
    if( BCP() = **Unsat** )
      $\beta$ := CONFLICT-ANALYSIS($C_{DB}$,trail)
      if( $\beta$ < user-level):
        return **Unsat**
      else:
        Backtrack($C_{DB}$, trail, $\beta$)
        sat-level := $\beta$
        continue
    theoryEngine.check(StandardEffort)
    if( there are enqueued $\mathcal{T}$-lemmas ):
      import $\mathcal{T}$-lemmas as clauses into $C_{DB}$
      //may require Conflict-Analysis/Backtrack
      continue
    theoryEngine.propagate()
    if( there are enqueued $\mathcal{T}$-propagations ):
      push $\mathcal{T}$-propagated literals onto trail
      continue
    if(AllClausesAreSat($C_{DB}$, trail)):
      theoryEngine.check(FullEffort)
      if( there are enqueued T-lemmas ):
        continue
      return **Sat**
    $[x \to v]_d$ := PickBranch(clauses, trail)
    sat-level := sat-level + 1
    push $[x \to v]_d$ onto trail
```

Figure 1.5: DPLL($\mathcal{T}$) extended CDCL loop
Psuedocode for a DPLL($\mathcal{T}$) extended CDCL loop.

```
class TheorySolver {
  void assert(Literal p);
  void check(effort);
  void addSharedTerm(Term t);
  void preregister(Atom p);
  void propagate();
  void explain(Propagated Literal p);
};
```

Figure 1.6: Abstraction of a Theory Solver.

lemma if the assertions are inconsistent or a splitting lemma to refine an unknown state.

Theory solvers may be directed to propagate theory-entailed literals. CVC4 currently restricts the propagated literals to be over atoms the PROPENGINE or THEORYENGINE know about. The THEORYENGINE tells the theory solvers which atoms are known to the PROPENGINE using T-PREREGISTER(p). To better support sharing the THEORYENGINE supports propagating equalities and disequalities between shared terms. If t is shared, the THEORYENGINE calls `TheorySolver::addSharedTerm(t)`. Once t and s are marked as shared, the theory solver may propagate $s = t$ or $s \neq t$ if it is entailed. A propagated literal p is sent out on the output stream and is then handled by THEORYENGINE. The theory solver may be asked to explain any of the propagations it has made in the current sat-level using T-EXPLAIN(p).

1.5.7 Context Levels and Backtracking

A key concept in developing DPLL($\mathcal{T}$) SMT solvers is the notion of *context dependence*. Most system components are designed to be consistent with respect to a given set of assertions. The two major sets of assertions are the user assertions $\mathcal{A}_U$ and the SAT solver's set of assertions `trail` (which are sent to the theory solvers). Both sets of assertions are implemented as stacks which implicitly represent conjunctions. Assertions are added to the tops of these stacks in the current context-level. Increasing the context-level is done by an explicit *push* of the context level. On a push, the sizes of the assertion lists is saved. The context-level is decreased by an explicit *pop* operation. On a pop, the top of the stack of assertions is popped until the size of the stack matches the size that was saved on the previous push.

SMT solvers are designed to "live on the stack." Their datastructures are designed to be either: dependent upon sat level, dependent on the user context-level, local to a function call or global. Datastructures that are dependent on the context-level are automatically backtracked to a previous state. They make and save incremental amounts of work in response to new assertions that they then lose when the assertion is popped. This helps make the solvers more incremental, efficient, and in many cases easier to reason about.

1.5.8 Preprocessing

Preprocessing passes transform a set of assertions, $\mathcal{A}$, with the goal of making them more efficiently solvable. What preprocessing passes are allowed to do is quite broad: the only theoretical limit on what preprocessing passes can do is that the resulting $\mathcal{A}'$ is equisatisfiable and that handles for $\mathcal{A}'$ can be converted into handles for $\mathcal{A}$ [41]. Examples of preprocessing include:

- Learning that a literal ℓ is entailed by $\mathcal{A}$ (much like BCP in SAT).
- If an equality $x = t$ is entailed by $\mathcal{A}$ and t does not contain x, then t may be substituted for x everywhere in $\mathcal{A}$.
- If a term such as $x + t$ is unconstrained, e.g. x is a variable that only appears in $x + t$, then $x + t$ may be replaced by a fresh variable y.
- The theory may add clauses that are likely to speed up the propositional search, e.g. $(x \leqslant 2) \implies (x \leqslant 5)$.
- Attempt to solve a bitvector problem using a smaller bit-width and see if the satisfying answer scales.

The eager style reasoning discussed in Sec. 1.4 is done only by preprocessing passes in CVC4. Section 2.1.2 discusses theory preprocessing for arithmetic.

1.5.9 Theory Sanitizing

The input language that SMT solvers are capable of supporting may not directly correspond to the language the theory solvers can efficiently support. Theory sanitizing passes reduce these more complex expressions into terms that the theory can support. These new constraints may require adding fresh new theory variables that do not appear in the original formula, and adding new constraints to ensure that the proper semantics of the more complex expression are satisfied. These new variables are often called *skolems* after Thoralf Skolem. The result of sanitizing is an equisatisfiable formula. Equisatisfiable transformations often introduce new variables that are restricted to take on the value of more complex expressions. These new variables are often substituted for the original expressions. We write $s[t/u]$ to denote the result of replacing the term t with the term u whenever it appears in the term s.[9] Examples of sanitizing include:

- replacing non-Boolean ite terms in $\mathcal{A}$ with a skolem s,

$$\mathcal{A}[(\mathsf{ite}\ c\ t\ e)/s] \wedge (\mathsf{ite}\ c\ (=\ s\ t)\ (=\ s\ e))$$

- removing all instances of a user defined non-recursive macro function $m = \lambda x.e$,

$$\mathcal{A}[(m\ t)/e[x/t]]$$

- and replacing function symbols with internally defined macros.

Expanding internal macro functions allows for solving some particularly difficult cases that may even require two theories. The primary motivation of internally defined macros is to handle division-by-zero. The SMT-LIB standard requires assigning a total function to the function symbol / which includes assigning $(/\ t\ 0)$ a real value for all t. CVC4 handles this by replacing $(/\ t\ s)$ with

$$\big(\mathsf{ite}\ (=\ s\ 0)\ (/_{\mathsf{by-0}}\ t)\ (/_{\mathsf{total}}\ t\ s)\big)$$

sending $(/_{\mathsf{by-0}}\ t)$ to the uninterpreted function theory as a shared term, and otherwise using a version of division $/_{\mathsf{total}}$ that CVC4 can safely make total arbitrarily.

1.5.10 Nodes

Formulas, terms, and sorts are all built on top of the Node datastructure in CVC4. Nodes form a Directed Acyclic Graph (DAG). The leaves of the DAG are variables and constants. Nodes representing applications of built-in functions simply contain a list of children. Every Node has a *Kind* that describes what each Node is. If a Node has a Kind indicating that it is a variable, then the rest of the node is a unique unsigned integer. Every datastructure that is a constant has a unique kind associated with it (The Kind also indicates a unique C++ type for this constant.) Built-in theory function symbols and logical connectives each have their own kind. Uninterpreted functions have the kind APPLY and take the first child as the function being applied.[10]

Nodes are structurally identical if they are built from the same sequence of function applications over the same variables and constants. To indicate that

[9] We are ignoring variable capture during substitution for simplicity.

[10] Similar constructs to uninterpreted functions that cannot be built into the language, such as constructors and selectors for the theory of datatypes, are built using analogs of APPLY.

$$\frac{\neg\textbf{true}}{\textbf{false}}\ \textsc{Not-True} \qquad \frac{\neg\textbf{false}}{\textbf{true}}\ \textsc{Not-False}$$

$$\frac{\textbf{true} \vee \phi}{\textbf{true}}\ \textsc{Disj-True} \qquad \frac{\textbf{false} \vee \phi}{\phi}\ \textsc{Disj-False}$$

$$\frac{\textbf{true} \wedge \phi}{\phi}\ \textsc{Conj-True} \qquad \frac{\textbf{false} \wedge \phi}{\textbf{false}}\ \textsc{Conj-False}$$

$$\frac{\textbf{true} = \phi}{\phi}\ \textsc{Eq-True} \qquad \frac{\textbf{false} = \phi}{\neg\phi}\ \textsc{Eq-False}$$

Figure 1.7: Example rewrite rules
Example Rewrite Rules.

two Nodes s and t are structurally equal, we write $s \approx t$. The memory associated with Nodes belongs to a NodeManager. The NodeManager is responsible for controlling the pool of Nodes. Upon the creation of nodes the NodeManager uses hash-consing to ensure that if $s \approx t$, then the Nodes are also equal up to pointer equality. The NodeManager used throughout a given SmtEngine is global for efficiency and clarity. Nodes use reference counting for garbage collection, and the NodeManager periodically reclaims zombie nodes (nodes without an incoming edge). Cycles in the Node graph cannot happen as Nodes form a DAG. The user-exposed Expr class combines a Node with a pointer to the NodeManager that owns the Node.

1.5.11 Rewriting

The REWRITER procedure performs $\mathcal{T}$-valid term transformations. These are simple equational rules that are frequently invoked to create simpler terms and simplify the number of cases handled by other components.

For example, the simple rewriting rules shown in Fig. 1.7 allow for removing the constants **true** and **false** from a set of clauses. Each of these rules allows for either removing an instance of **true** or **false** or reducing the height of the Node. All of the rules for binary connectives need to be applied up to commutativity, i.e. both $\phi = \textbf{true}$ and $\textbf{true} = \phi$ should trigger EQ-TRUE. Exhaustive application of these rules reduces a propositional formula to either **true**, **false** or a formula not containing **true** or **false**. These rules also preserve CNF. CVC4's rewriter has three main properties. The most important is that all its transformations are theory valid,

$$\models_{\mathcal{T}} t = \textsc{Rewriter}(t).$$

Next, the rewriter is *idempotent*. Applying the rewriter to the result of a rewritten node returns the same node,

$$\textsc{Rewriter}(t) \approx \textsc{Rewriter}(\textsc{Rewriter}(t)).$$

To denote that a term t rewrites to a term s, we write

$$t \xrightarrow{\textsc{Rewriter}} s \text{ as shorthand for } \textsc{Rewriter}(t) \approx s.$$

A rewriter is strongly normalizing over a set of terms S if for all $s, t \in S$, there are terms $s', t' \in S$ such that $t \xrightarrow{\textsc{Rewriter}} t'$, $s \xrightarrow{\textsc{Rewriter}} s'$ and

$$\models_{\mathcal{T}} s = t \quad \text{iff} \quad t' \approx s'.$$

The rewriter is required to be strongly normalizing over the set of terms built over [interpreted] constants by interpreted function application, i.e. the rewriter should evaluate interpreted functions over constants to constant terms. Beyond constant evaluation, rewrites can be used for a number of purposes:

- Rewrites may be used to simplify formulas so cases do not constantly have to be reimplemented. For example, reducing $\vee$ containing **true** to **true** or $\wedge$ chains that contain both x and $\neg x$ to **false**, etc.

- Rewrites can eliminate convenient but unnecessary symbols such as unary negation or division by a non-zero rational constant q.

$$-u \rightarrow -1 \cdot u$$
$$(\ /\ s\ q) \rightarrow \left(\frac{1}{q}\right) \cdot u$$

- More powerful rewrites can increase DAG sharing by simplifying many equivalent formulas to the same node.

$$(x + y + z) = (y + (x + z)) = (x + x + y + y + 2 \cdot z)/2$$

- Even more extensive rewrites can put a theory's atoms into a normal form. Atoms in a normal form are logically equivalent iff syntactically equal.

$$\textsc{Rewriter}(t) \not\approx \textsc{Rewriter}(s) \implies t \neq s \text{ is satisfiable}$$

CVC4's rewriter is cached so that repeated calls to the rewriter stay relatively inexpensive. Calling the rewriter on a node that has already been rewritten should be roughly as expensive as one hashtable lookup.

1.5.12 If-then-else Term Removal

A straightforward procedure for removing if-then-else terms is to replace each ite term with a fresh Skolem variable. Let $(\mathrm{ite}\ c\ t\ e)$ be an if-then-else term with sort S in ϕ, and let s be a fresh variable of sort S that does not appear in ϕ. Let ϕ' be the result of replacing $(\mathrm{ite}\ c\ t\ e)$ with s,

$$\phi' \equiv \phi[(\mathrm{ite}\ c\ t\ e)/s]$$

then ϕ and $\phi' \wedge (s = (\mathrm{ite}\ c\ t\ e))$ are equisatisfiable. By taking this one step further and *lifting* the ite above the equality in $s = (\mathrm{ite}\ c\ t\ e)$, we get that

$$\phi \quad \text{and} \quad \phi' \wedge (\mathrm{ite}\ c\ (=\ s\ t)\ (=\ s\ e)) \text{ are equisatisfiable} .$$

By applying this transformation to all non-Boolean ite terms in a bottom-up fashion, this results in an equisatisfiable formula that does not contain any non-Boolean ite terms in linear time. Boolean ite terms can be directly supported during CNF conversion. (See Section 2.1.2 for a description of an analogous equisatisfiable reduction.)

Chapter 2

Simplex for Satisfiability Modulo Theories

Theory solvers for quantifier-free linear real arithmetic take as input comparisons of linear terms. For example, all of the following are atoms in quantifier free real arithmetic:

$$x = 5 + z, \quad z - y \geqslant 1, \quad -\frac{1}{5}x - y \geqslant 0, \quad y > -1. \tag{2.1}$$

The terms of linear arithmetic are sums of rational constants and real variables multiplied by rational constants, e.g. $\frac{1}{5}x + y$, 0, x, $5 + z$, etc. Formulas in Quantifier-Free Linear Real Arithmetic (QF_LRA) are Boolean combinations of atoms of the form $s \bowtie t$ for linear terms s and t and $\bowtie \in \{=, <, >, \leqslant, \geqslant\}$. The theory solver decides whether or not there is an assignment of the variables into the reals $\mathbb{R}$ that makes all of the input assertions true. In (2.1), the first 3 constraints can be satisfied by the assignment $[y \to -1]$, $[z \to 0]$ and $[x \to 5]$. Including the fourth constraint, $y > -1$, makes the system unsatisfiable.

The theory of reals $\mathcal{T}_{\mathbb{R}}$ used in SMT solvers is a straightforward extension of the theory of real closed fields [65, Section 2.2]. The signature $\Sigma_{\mathbb{R}}$ for the theory of the reals starts with the core function symbols $\langle 0, 1, +, \cdot, < \rangle$. These function symbols are interpreted by the real closed field axioms, but as $\mathcal{T}_{\mathbb{R}}$ is complete (Section 2.1.1), it will be enough for our purposes to assume these symbols have their standard interpretation over the real numbers. Instead of addressing the full language of the theory, the work of this thesis focuses on decision procedures for a subset of this language, quantifier-free linear arithmetic.

The simplex algorithm introduced by Dutertre and de Moura in [46] for use in the DPLL($\mathcal{T}$) framework is the core reasoning module for linear arithmetic in nearly every state-of-the-art Satisfiability Modulo Theories (SMT) solver. The algorithm—which we will call SIMPLEXFORSMT—searches for either a satisfying model or a conflict. This algorithm has many attractive properties within the context of DPLL($\mathcal{T}$). Implementations are naturally incremental. The conflicts generated by the decision procedure are minimal. The search tends to be efficient on representative benchmarks for both satisfiable and unsatisfiable queries. It is relatively easy to create an initial implementation. SMT solvers that have implemented this algorithm include Barcelogic [18], CVC4 [7], MathSAT [27], OpenSMT [23], SMTInterpol [26], Yices [48], Yices 2 [45], and Z3 [38].

This chapter describes many aspects of implementing the SIMPLEXFORSMT algorithm that have not been discussed in the published literature. Many of these aspects are commonly known among the restricted community of implementers of this algorithm, but have so far remained unpublished. Section 2.1 contains background, the preprocessing required for the algorithm, the core invariants of the algorithm and an abstract description of the algorithm. This discussion is intended to be an introduction for those unfamiliar with the algorithm. The second half of this chapter (Section 2.2) elaborates on various aspects of the algorithm and discusses how to efficiently implement a theory solver for linear arithmetic. Each aspect of the abstract algorithm is refined and expanded in its own section and will rely on an understanding of the first section. This second half describes an improved method for detecting conflicts earlier and more efficiently over all candidates. This improved detection method leads to an amortized constant amount of additional work. A new algorithm for strengthening the detected conflicts is presented. Additionally, this chapter gives a minor contribution by describing how to split disequalities in a lazy fashion and compute models with disequalities. By way of example, a number of implementation details of CVC4's theory solver for `QF_LRA` are given.

2.1 Abstract Description

The SIMPLEXFORSMT algorithm is a decision procedure for the satisfiability of conjunctions of literals in the `QF_LRA` logic. This section describes the algorithm beginning with a formal description of `QF_LRA` in 2.1.1, then continuing with preprocessing in 2.1.2, invariant and subroutines in 2.1.3, and concluding with the simplex algorithm in 2.1.4.

2.1.1 Quantifier-Free Linear Real Arithmetic

The set of variables $\mathcal{X}$ will be a set containing only variable symbols belonging to the theory, those labeled with the sort `Real`, $\{x_1 : \texttt{Real}, x_2 : \texttt{Real}, \ldots, x_n :$

$\texttt{Real}\}$ where each x_i is uniquely indexed by an integer i the range 1 to n.[1] The signature of the theory of reals $\Sigma_{\mathbb{R}}$ is classically $\langle 0, 1, +, \cdot, < \rangle$. For notational compactness, the signature is also extended to include all rational constants. *Linear terms* are the minimal inductive set generated from $\mathcal{X}$ and the rational constants by the function symbols $+$ and $\cdot$ where $\cdot$ is restricted so that in each application either s or t is a rational constant in $s \cdot t$. For convenience, the signature is extended with additional comparison operations $\leqslant$, $>$, and $\geqslant$ and division / (with linear terms including division by non-zero rational constants). The atoms of $\texttt{QF\_LRA}$ are of the form

$$(\bowtie\ \ s\ t) \text{ for } \bowtie \in \{=, <, >, \leqslant, \geqslant\}$$

and linear terms s and t. The semantics of the functions in $\Sigma_{\mathbb{R}}$ is fixed by the real closed field axioms [65]. The atoms of this logic can always be rewritten to atoms of the form $\sum_{x_j \in \mathcal{X}} c_j x_j \bowtie d$, where c_j, d are rational constants, $\bowtie \in \{=, \leqslant, \geqslant\}$. The input problem may contain both Boolean variables and if-then-else terms over linear terms; however, both of these are either handled by the SAT solver or are removed during preprocessing and rewriting. Neither appear in the assertions sent to the theory solver.

While this chapter focuses on quantifier-free linear real arithmetic, algorithms for the full theory of real arithmetic including both non-linear arithmetic and quantifiers exist. Tarski showed that the theory of real closed fields admits quantifier elimination and is decidable [106]. Collins' Cylindrical Algebraic Decomposition method improved Tarski's non-elementary quantifier elimination procedure to doubly exponential in the number of quantifier alternations [31]. Jovanović and de Moura recently gave an effective method for quantifier-free non-linear real arithmetic based on a combination of the Model-Constructing Calculus [40] and partial Cylindrical Algebraic Decomposition [70].

2.1.2 Preprocessing

There are many well known equivalent ways of formulating linear systems. (See [56, Chapter 7].) We will use one that optimizes the efficient addition and removal of constraints during DPLL($\mathcal{T}$) search. As a preprocessing pass over an input formula ϕ, inequalities with at least 2 variables are transformed by introducing a fresh real variable x_i for each unique left hand side $\sum c_j x_j$ appearing in ϕ. We call the fresh variable x_i an *auxiliary* variable. The *structural* variables are those that appear in the original input formula ϕ. To denote that a variable x_i is auxiliary, we write that $i \in \text{Aux}$ where Aux is the set of all auxiliary vari-

[1] $\mathcal{X}$ will be extended dynamically throughout the algorithm. Thus is it is conceptually treated as a countable enumerable set with only the first n elements enumerated at any given time.

ables. As each auxiliary variable x_i is fresh, it is equisatisfiable to restrict that x_i is always equal to $\sum c_{i,j}x_j$. An interpretation M that satisfies ϕ can be extended to an interpretation M' that satisfies both ϕ and the new equalities,

$$M \models_{\mathbb{R}} \phi \implies M' \models_{\mathbb{R}} \phi \wedge \bigwedge_{i \in \text{Aux}} x_i = \sum c_{i,j}x_j$$

$$\text{where } M' = M\left[x_i \to \text{Eval}\left(M, \sum c_{i,j}x_j\right) \middle| i \in \text{Aux}\right]$$

And clearly, if M' is any interpretation that satisfies $\phi \wedge \bigwedge x_i = \sum c_{i,j}x_j$, then it satisfies ϕ. Let ϕ' be the result of replacing in ϕ each appearance of $\sum c_{i,j}x_j$ with auxiliary variable x_i. Then, if M is an interpretation that satisfies ϕ and $\bigwedge x_i = \sum c_{i,j}x_j$, then M also satisfies ϕ' and vice versa.

$$M \models_{\mathbb{R}} \phi \wedge \bigwedge_{i \in \text{Aux}} x_i = \sum c_{i,j}x_j \iff M \models_{\mathbb{R}} \phi' \wedge \bigwedge_{i \in \text{Aux}} x_i = \sum c_{i,j}x_j.$$

The formulas ϕ and ϕ' are equivalent modulo $\mathcal{T}_{\mathbb{R}}$ and $\bigwedge x_i = \sum c_{i,j}x_j$.

We maintain a copy of the sum terms that define each auxiliary variable. This is implicitly encoded in an $n \times n$ matrix A where the i'th row (A_i) represents the equality $x_i = \sum c_{i,j}x_j$. For the auxiliary variable x_i, the coefficient of $A_{i,i}$ is -1 and the coefficient for $j \neq i$ is $A_{i,j} = c_{i,j}$ from the defining sum. This implicitly makes $A_{i,k} = 0$ for x_k not appearing in the equality and the row $A_j = 0$ for $j \notin \text{Aux}$.

Assume for now that all of the equalities in ϕ' are rewritten using

$$x = d \iff (x \leqslant d \wedge x \geqslant d).$$

(See Section 2.2.1 for handling strict inequalities.) By applying these simple transformations, the input formula ϕ has been transformed into a formula

$$\psi \equiv \phi'' \wedge \bigwedge_{i \in \text{Aux}} x_i = \sum c_{i,j}x_j \tag{2.2}$$

such that ϕ and ψ are equisatisfiable and ϕ'' contains only inequalities $\leqslant, \geqslant$ over a single variables. Then, by invoking the SMT solver on ψ, all of the constraints sent to the theory solver are either special equalities defining an auxiliary variable $x_i = \sum c_{i,j}x_j$ or an inequality over a single variable. Let us also assume that all of the inequalities sent to the theory are non-strict i.e. $x \leqslant y$ but not $x > 0$. (See Section 2.2.1 for handling strict inequalities.) The set of assertions, $\mathcal{A}$, sent to the theory solver is then always equivalent to

$$\bigwedge_{i \in Aux} x_i = \sum c_{i,j} x_j \wedge \bigwedge x_j \geqslant d_j \wedge \bigwedge x_k \leqslant d_k$$

where $\bigwedge x_j \geqslant d_j$ and $\bigwedge x_k \leqslant d_k$ are finite conjunctions of lower and upper bounds on variables. At most one lower and at most one upper bound needs to be kept per variable, as by transitivity, the strongest bound on variable x entails the others.

$$\frac{c \leqslant d}{x \leqslant c \models_{\mathbb{R}} x \leqslant d} \quad \frac{c \geqslant d}{x \geqslant c \models_{\mathbb{R}} x \geqslant d}$$

Propagation of this form is sometimes called *unate* propagation [47]. We further use $l(x)$ and $u(x)$ to denote the strongest lower and upper bounds on a specific variable x. If x has no lower (upper) bound, then $l(x) = -\infty$ ($u(x) = +\infty$). By removing weaker bounds and implicitly allowing extended comparisons[2], we can write the input to the theory solver as a conjunction of equalities defining auxiliary variables and a single upper and lower bound for every variable:

$$\bigwedge_{i \in Aux} x_i = \sum A_{i,j}\, x_j \wedge \bigwedge_{x_i \in \mathcal{X}} l(x_i) \leqslant x_i \leqslant u(x_i). \tag{2.3}$$

The theory solver searches for an assignment $a : \mathcal{X} \mapsto \mathbb{R}$ that satisfies the constraints in (2.3).

As shorthand, we will adopt linear algebra notation and conventions to denote elements of $\mathcal{X}$, l, u, a and A and their relationships. The set of variables $\mathcal{X}$ can be viewed as the n-dimensional column vector of variable symbols $\langle x_1, \ldots, x_n \rangle$. For all $i \in [1, n]$, we often write l_i as a synonym for $l(x_i)$. Similarly, $u_i = u(x_i)$ and $a_i = a(x_i)$. For each $i \in$ Aux, the constraint $x_i = \sum A_{i,j}\, x_j$ that defines x_i can be represented as the dot-product of row i in A and the vector of the variables, $A_i \cdot \mathcal{X} = 0$. Combining these row constraints together for all auxiliary variables, we can compactly write

$$A\,\mathcal{X} = 0 \quad \text{as shorthand for} \quad \bigwedge_{i \in Aux} x_i = \sum A_{i,j}\, x_j$$

The assertions sent to the theory solver can be written as

$$A\,\mathcal{X} = 0 \wedge l \leqslant \mathcal{X} \leqslant u, \tag{2.4}$$

and a satisfying assignment a must satisfy $A\,a = 0$ and $l \leqslant a \leqslant u$.[3]

[2] For all $x \in \mathbb{R}$ it is the case that $-\infty < x < +\infty$.

[3] The dot product $A_i \cdot \mathcal{X}$ is in the polynomial ring $\mathbb{R}[x_1, \ldots, x_n]$, i.e. it is still a symbolic

```
1: procedure UPDATE($j, \Delta$)
2:     $a_j \leftarrow a_j + \Delta$
3:     for all $i$ such that $T_{i,j} \neq 0$ do
4:         $a_i \leftarrow a_i + T_{i,j}\Delta$
```

Figure 2.1: Update changes a_j by Δ for $j \in \mathcal{N}$.

2.1.3 Invariants and Subroutines

The set of constraints $A\,\mathcal{X} = 0$ over the implicit matrix A is implemented by a concrete data structure, the *tableau* T. The matrix T is an $n \times n$ matrix in *tableau form*. We will refer to the entry in row i and column j of T as $T_{i,j}$, and use T_i to denote the i-th row of T. The variables $\mathcal{X}$ are partitioned into *basic* variables and *non-basic* variables. The indices of the basic and non-basic variables are the sets $\mathcal{B}$ and $\mathcal{N}$. A matrix is in tableau form if for each column i such that $i \in \mathcal{B}$, we have $T_{k,i} = 0$ for all $k \neq i$ and $T_{i,i} = -1$, and for $j \in \mathcal{N}$, the row T_j all zeroes. Each nonzero row T_i of T represents a constraint $x_i = \sum_{j \in \mathcal{N}} T_{i,j} x_j$. The basic variables are initially exactly the auxiliary variables with $T = A$. One may intuitively think of the basic variables in the tableau as the "solved for" variables in Gaussian elimination.[4]

The simplex solver in SIMPLEXFORSMT makes a series of changes to an initial assignment a and the tableau T until the constraints are satisfied or determined to be unsatisfiable. Throughout this search, five main invariants are maintained.

(I1) $l \leqslant u$: The upper and lower bounds of each variable are consistent, i.e. for all $x_i \in \mathcal{X}$, $l_i \leqslant u_i$.

(I2) $l_j \leqslant a_j \leqslant u_j$ for all $j \in \mathcal{N}$: The assignment to each non-basic variable is within its bounds.

(I3) $Ta = 0$: The assignment a satisfies the linear equalities in the tableau T.

(I4) $T_i = \sum y_j\, A_j$ for some $y \in \mathbb{R}^n$: Every row in T is equal to a sum of rows in A scaled by some real values y_j.

(I5) The matrix T is in tableau form.

The procedures in this subsection update a, T, l and u while maintaining (I1)-(I5).

expression, while $A_i \cdot a$ is a real number. Similarly, $l \leqslant \mathcal{X} \leqslant u$ are symbolic comparisons while $l \leqslant a \leqslant u$ are [extended-]numerical comparisons.

[4] This is slightly incorrect. See section 2.2.7 for more on the tableau.

```
Require: $T_{i,j} \neq 0, i \neq j$
1: procedure PIVOT$(i, j)$
2:     $T_j \leftarrow T_j - \frac{1}{T_{i,j}} T_i$
3:     for all $k$ such that $T_{k,j} \neq 0 \wedge k \neq j$ do
4:         $T_k \leftarrow T_k + T_{k,j} T_j$
5:     $\mathcal{B} \leftarrow \mathcal{B} \setminus \{i\} \cup \{j\}, \mathcal{N} \leftarrow \mathcal{N} \setminus \{j\} \cup \{i\}$
```

Figure 2.2: An abstracted pivot of x_i and x_j ($T_{i,j} \neq 0$).

The invariant (I3) states that the assignment satisfies the tableau T can be compactly written as $Ta = 0$. To initially satisfy this invariant, one can set $a_j \leftarrow 0$ for all $x_j \in \mathcal{X}$. Consider trying to update the assignment of a single non-basic variable x_j by an amount Δ. A simple way to update a_j while maintaining (I3) is to also update the assignments of all of the basic variables x_i that depend on x_j, i.e. where $T_{i,j} \neq 0$. If a is the current assignment, then the following holds on x_i's row.

$$a_i = T_{i,j} a_j + \sum_{k \notin \{i,j\}} T_{i,k} a_k \tag{2.5}$$

If a' is the next assignment with $a'_j = a_j + \Delta$ and if all of the other non-basic variables are unchanged ($a'_k = a_k$), then in order for (I3) to hold, the next assignment of x_i must satisfy:

$$\begin{aligned} a'_i &= T_{i,j} a'_j + \sum_{k \notin \{i,j\}} T_{i,k} a'_k = T_{i,k}\Delta + T_{i,j} a_j + \sum_{k \notin \{i,j\}} T_{i,k} a_k \\ &= T_{i,j}\Delta + a_i \text{ [by (2.5)].} \end{aligned}$$

This new assignment a' then satisfies (I3). Figure 2.1 gives an algorithm UPDATE that updates the assignment of a non-basic variable in this fashion. If the invariant (I2) that non-basic variables are within bounds is initially true, then $l_j \leqslant a_j \leqslant u_j$ remains true if changing the assignment of x_j by Δ remains in bounds,

$$l_j \leqslant a_j + \Delta \leqslant u_j.$$

Changes to the tableau T are introduced via *pivoting*. An abstract algorithm for pivoting is given in Figure 2.2. The essence of pivoting is to look at the row of a basic variable x_i and a non-basic variable x_j on row i (where $T_{i,j} \neq 0$), solve row i for x_j and to use this "new equality" to substitute out x_j on other rows. After pivoting, x_j becomes basic and x_i becomes non-basic. Solving row i for x_j

```
Require: (I3), (I2), $T_{i,j} \neq 0$, $i \neq j$, and $l_i \leqslant a_i + \Delta \leqslant u_i$
Ensure: (I3), (I2)
1: procedure PIVOTANDUPDATE($i, j, \Delta$)
2:     PIVOT($i, j$)
3:     UPDATE($i, \Delta$)
```

Figure 2.3: A composed pivot and update operation.

is equivalent to multiplying it by $-\frac{1}{T_{i,j}}$.

$$x_i = T_{i,j}x_j + \sum_{k \notin \{i,j\}} T_{i,k}x_k \quad \Longrightarrow \quad x_j = \frac{1}{T_{i,j}}x_i + \sum_{k \notin \{i,j\}} -\frac{T_{i,k}}{T_{i,j}}x_k$$

Substitution of x_j with the right hand side on another row m is done by simple row addition. The new row T_j is multiplied by $T_{m,j}$ and added to T_m to cancel out the x_j term.

$$\begin{array}{rlllllll}
T_j: & T_{m,j}(& & -x_j & +\frac{1}{T_{i,j}}x_i & +\sum & -\frac{T_{i,k}}{T_{i,j}}x_k &)=0 \\
T_m: & + & -x_m & +T_{m,j}x_j & & +\sum & T_{m,k}x_k & =0 \\
\hline
T'_m: & & -x_m & & +\frac{T_{m,j}}{T_{i,j}}x_i & +\sum & \left(-\frac{T_{m,j}T_{i,k}}{T_{i,j}} + T_{m,k}\right)x_k & =0
\end{array}$$

Applying these operations across all rows (except T_j) generates a new matrix T'. The new T' is in tableau form with x_j being basic, x_i being non-basic and other variables remaining basic or non-basic as they were. The new T' maintains (I4) as only scaled row additions are used to generate new rows. As the assignment a is in the null space of the previous tableau ($Ta = 0$), a must also satisfy $T'a = 0$ and (I3) holds.

The main operation of simplex is to swap a basic variable x_i that currently does not satisfy its bounds, either $a_i > u_i$ or $a_i < l_i$, with a non-basic variable x_j on x_i's row. This operation can be done by composing a pivot between x_i and x_j, PIVOT(i, j), and an update to the newly non-basic x_i by an amount Δ, UPDATE(i, Δ). As (I2) is violated between these two calls, we make the composition of these two explicit by a procedure PIVOTANDUPDATE (Fig. 2.3), and require that PIVOTANDUPDATE reestablishes both invariants. This is possible whenever $l_i \leqslant a_i + \Delta \leqslant u_i$.

The previous procedures have so far not manipulated the bounds in l and u. The algorithms in Figures 2.4 and 2.5 are used for changing l and u while maintaining (I1)-(I2).

As discussed in Section 1.4, the SAT solver will make propositional assignments to the atoms of the theory, i.e. it assigns $(x_i \leqslant c)$ to **true**. This is then

```
Require: (I1), (I3) and (I2)
Ensure: (I1), (I3) and (I2)
1: procedure AssertUpper($x_i \leqslant c$)
2:     if $c < l_i$ then
3:         return (Conflict $\{x_i \leqslant c, x_i \geqslant l_i\}$)
4:     else if $c < u_i$ then
5:         $u_i \leftarrow c$
6:         if $a_i > c \wedge i \in \mathcal{N}$ then
7:             Update($i, u_i - a_i$) // Set $a_i$ to $u_i$
```

Figure 2.4: Pseudocode for asserting an upper bound.

```
Require: (I1), (I2) and (I3)
Ensure: (I1), (I2) and (I3)
1: procedure AssertLower($x_i \geqslant c$)
2:     if $c > u_i$ then
3:         return (Conflict $\{x_i \geqslant c, x_i \leqslant u_i\}$)
4:     else if $c > l_i$ then
5:         $l_i \leftarrow c$
6:         if $a_i < c \wedge i \in \mathcal{N}$ then
7:             Update($i, l_i - a_i$) // Set $a_i$ to $l_i$
```

Figure 2.5: Pseudocode for asserting a lower bound

sent to the theory solver. Assuming that the preprocessing described in Section 2.1.2 was run, the incoming literal will either be an upper bound ($x \leqslant c$) or a lower bound ($x \geqslant c$) on a variable. When a new upper bound $x_i \leqslant c$ comes in, the procedure ASSERTUPPER is called to incrementally update u_i. (A similar procedure ASSERTLOWER handles the constraints $x_i \geqslant c$.) To ensure (I1) is maintained, ASSERTUPPER first checks if $c < l_i$. If it is, then the two constraints $l_i \leqslant x_i$ and $x_i \leqslant c$ are unsatisfiable, and this is reported as a conflict, and the procedure exits. Otherwise, if $c \geqslant u_i$, the bound $x_i \leqslant c$ holds whenever $x_i \leqslant u_i$, and $x_i \leqslant c$ can be ignored. In the final case where $l_i \leqslant c < u_i$, c is the stronger upper bound and u_i is updated to c. The procedure then checks if $a_i > c$ when x_i is non-basic. If it is, the assignment of x_i is updated to be equal to the new upper bound (UPDATE$(i, u_i - a_i)$). This ensures that (I2) still holds. The analogous ASSERTLOWER is the same up to the direction of the comparisons.

The invariants (I1)-(I3) smoothly interact with SAT-context-level pushes and pops. The lower and upper bounds on variables come in as a dynamic stream of new assertions $\mathcal{A}$. The values of l_i and u_i are always updated to be the strongest values in the current sat-context. This means that they can be implemented as SAT-context-level maps (where the value pointed to automatically backtracks to its value before the push). To distinguish between these datastructures pre or post SAT-context-level-pop, we will denote these as $l_{pre-pop}$ and $l_{post-pop}$, etc.

Initially, no bounds are asserted and $l(x_i) = -\infty$ ($u(x_i) = +\infty$). Notice that the bound being popped back to is always weaker than the bound being popped back from,

$$l_{post-pop}(x_i) \leqslant l_{pre-pop}(x_i) \text{ and } u_{pre-pop}(x_i) \leqslant u_{post-pop}(x_i).$$

Suppose that T and a are left unchanged by context-level pushes and pops. The above inequalities then ensure that the main invariants are maintained on pops.

Lemma 2.1. *If (I1)-(I5) hold pre-SAT-context-level-pop for* $l_{pre-pop}$, $u_{pre-pop}$, a *and* T, *then (I1)-(I5) hold post-SAT-context-level-pop for* $l_{post-pop}$, $u_{post-pop}$, a *and* T.

SAT-context-level pushes trivially maintain (I1)–(I5) as T, a, l, and u remain unchanged.

2.1.4 An Abstract Theory Check

We now give the algorithm for the SIMPLEXFORSMT decision procedure in Fig. 2.6. After the preprocessing described in Section 2.1.2, the algorithm decides the satisfiability of assertions to the theory solver of the form:

$$T\mathcal{X} = 0 \wedge l \leqslant \mathcal{X} \leqslant u.$$

```
procedure AbsSimplexForSMTCheck
    loop
        if (Conflict C) $\in$ outputStream then
            return (Conflict C)
        else if $E = \varnothing$ then
            return (Sat $a$)
        else
            $\sigma \leftarrow$ SimplexForSMTSelect()
            if $\sigma$ is an update $\langle i, \Delta, j\rangle$ then
                PivotAndUpdate($x_i, x_j, \Delta$)
```

Figure 2.6: An abstract version of the main loop of the SimplexForSMT check procedure.

The algorithm will either find an assignment a that simultaneously satisfies all of the assertions or a minimal subset of the constraints $l \leqslant \mathcal{X} \leqslant u$, and $T\mathcal{X} = 0$ that is unsatisfiable.

The algorithm focuses on variables that do not currently satisfy their bounds. We say that x_i is an *error variable* if a violates one of the bounds on x_i, and we denote by E the set of indices of error variables, $E = \{i | a_i > u_i\} \cup \{i | a_i < l_i\}$. Because the assignment satisfies the rows of the tableau (I3), the assignment is satisfying whenever the set of error variables is empty ($E = \varnothing$). The invariant that non-basic variables are within bounds (I2) ensures that the error variables are basic, $E \subseteq \mathcal{B}$. Every round of SimplexForSMT does the following:

1. It checks for a conflict (**Conflict** C) on the output stream. If there is one, it returns the conflict (**Conflict** C). (How conflict generation is done is described later in the section.)

2. If there is no conflict, check if E is empty. If so, the current assignment satisfies the input assertions and (**Sat** a) is returned.

3. If the assignment is not satisfying, the subroutine in Fig. 2.7 selects a PivotAndUpdate operation to perform.

4. The proposed PivotAndUpdate operation is performed (if no conflict has been found).

The main loop in Fig. 2.6 repeats until either a conflict or a satisfying assignment is found. The burden of the algorithm lies in the selection subroutine given in Fig. 2.7. The routine first selects some basic variable x_i using the non-empty set of indices in E. Either x_i's assignment exceeds its upper bound ($a_i > u_i$) or it

```
Require: $E \neq \varnothing$
 1: procedure SELECT
 2:     select $i$ from $E$ // for the basic variable $x_i$ either $a_i > u_i$ or $a_i < l_i$
 3:     if $a_i > u_i$ then
 4:         entering $\leftarrow \left\{j \middle| T_{i,j} < 0, a_j < u_j\right\} \cup \{k|T_{i,k} > 0, a_k > l_k\}$
 5:         $\Delta \leftarrow u_i - a_i$
 6:     else // $a_i < l_i$
 7:         entering $\leftarrow \left\{j \middle| T_{i,j} < 0, a_j > l_j\right\} \cup \{k|T_{i,k} > 0, a_k < u_k\}$
 8:         $\Delta \leftarrow l_i - a_i$
 9:     if entering $\neq \varnothing$ then
10:         select $j$ from entering
11:         return $\langle i, \Delta, j \rangle$
12:     else
13:         if $a_i > u_i$ then
14:             $C \leftarrow$ UpperConflict($x_i$)
15:         else
16:             $C \leftarrow$ LowerConflict($x_i$)
17:         (Conflict $C$) $\rightarrow$ outputStream
18:         return NoOp
```

Figure 2.7: An abstract pivot and update selection routine.

violates its lower bound ($a_i < l_i$). As these two cases are symmetric, we will focus just on the $a_i > u_i$ case. The algorithm looks at the row of x_i for a non-basic variable x_j such that x_j is on the row T_i ($T_{i,j} \neq 0$). The assignment to x_i must be decreased in order for the assignment to satisfy $a_i \leqslant u_i$. If the coefficient for x_j is positive, $T_{i,j} > 0$, then decreasing the assignment to x_j decreases a_i. Similarly if $T_{i,j} < 0$, increasing x_j decreases a_i. To make progress towards a model where $a_i \leqslant u_i$, we have to be able to find an assignment where the assignment of one of these non-basic variables either increases or decreases in the appropriate direction. The notion of progress SIMPLEXFORSMT uses is quite weak–it can always make progress as long as x_i is not provably at a minimum using its row. Suppose that for all x_j with $T_{i,j} < 0$ (and $j \neq i$), we have $a_j = u_j$, and for all x_k with $T_{i,j} > 0$, we have $a_k = l_k$, then

$$\begin{aligned}
x_i &= \sum_{T_{i,j}<0, i\neq j} T_{i,j}x_j + \sum_{T_{i,k}>0} T_{i,k}x_k && \lhd\, T\mathcal{X} = 0 \\
&\geqslant \sum_{T_{i,j}<0, i\neq j} T_{i,j}u_j + \sum_{T_{i,k}>0} T_{i,k}l_k && \lhd\, \text{relax } x_j \text{ to } u_j \text{ and } x_k \text{ to } l_k \\
&= \sum_{T_{i,j}<0, i\neq j} T_{i,j}a_j + \sum_{T_{i,k}>0} T_{i,k}a_k && \lhd\, u_j = a_j \text{ and } l_k = a_k \\
&= a_i && \lhd\, Ta = 0
\end{aligned} \tag{2.6}$$

Thus we can infer that the variable x_i must be greater than or equal to its current assignment, $x_i \geqslant a_i$. This newly inferred lower bound is in conflict with the upper bound of x_i as $a_i > u_i$. We thus have the following conflict:

$$a_i > u_i,\ u_i \geqslant x_i \text{ and } x_i \geqslant a_i.$$

The set of constraints

$$\left\{x_j \leqslant u_j \middle| T_{i,j} < 0\right\} \cup \{x_k \geqslant l_k | T_{i,k} > 0\} \cup T_i \cdot \mathcal{X} = 0$$

is inconsistent and each constraint is entailed by the input assertions. This conflict can be seen as taking the equality $T_i\mathcal{X} = 0$, and replacing all variables x_j such that $T_{i,j} < 0$ with their upper bounds u_j (including x_i) and all x_k with l_k and inferring the inconsistency $0 > u_i - a_i \geqslant 0$. These constraints cannot yet be reported as a conflict. The equality $T_i \cdot \mathcal{X} = 0$ is entailed by $A\,\mathcal{X} = 0$, but $T_i \cdot \mathcal{X} = 0$ is not guaranteed to be an input assertion to the theory. In DPLL($\mathcal{T}$), reported conflicts must be in terms of assertions the SAT solver knows about in order to force the SAT solver to backtrack. The [non-minimal] theory-conflict, UpperConflict(x_i), is reported:[5]

$$\neg\left\{x_j \leqslant u_j \middle| T_{i,j} < 0\right\} \cup \{x_k \geqslant l_k | T_{i,k} > 0\} \cup A\,\mathcal{X} = 0$$

This conflict is reported on the output stream ((**Conflict** UpperConflict(x_i)) $\rightarrow$ `outputStream`). In this case, assume for consistency that the SIMPLEXFORSMT-SELECT returns a no-op candidate for PIVOTANDUPDATE. The main loop of Simplex detects the conflict and then stops on the next round. Now if x_i is not already at a minimum using the previous rule, then there is some x_j such that $T_{i,j} < 0$ and $a_j < u_j$, or an x_k such that $T_{i,k} > 0$ that $a_k > l_k$. SIMPLEXFORSMT then selects such a non-basic variable x_j (x_k) and tries to use this to fix a_i in one update and pivot operation by pivoting x_i with x_j and updating the assignment of x_i to be equal to its upper bound u_i,

$$\text{PIVOTANDUPDATE}(i, \Delta, j) \text{ with } \Delta = u_i - a_i.$$

The procedure ABSSIMPLEXFORSMTCHECK is sent the candidate triple $\langle i, \Delta, j\rangle$.

The case where $a_i < l_i$ is exactly the same as the previous case [up to a few sign flips]. The variable x_i can be proven to be at a maximum if for all x_j with $T_{i,j} > 0$, we have $a_j = u_j$, and for all x_k with $T_{i,k} < 0$ (and $k \neq i$), we have $a_k = l_k$. If this is the case, we have the following conflict (defined as LowerConflict(x_i))

$$\left\{x_j \leqslant u_j \middle| T_{i,j} > 0\right\} \cup \{x_k \geqslant l_k | T_{i,k} < 0\} \cup A\,\mathcal{X} = 0.$$

If this does not hold, a candidate x_j can be selected, and the pivot and update operation pivots x_i with x_j and updates the assignment of x_i to the violated lower bound.

SIMPLEXFORSMT is sound as it only terminates with correct answers: either (**Sat** a) or (**Conflict** C). If the algorithm is terminating, it is also complete. The algorithm becomes terminating by imposing an additional simple variable selection criterion. Assume an arbitrary total order on the variables of $\mathcal{X}$, $\prec$. The additional criteria on Fig. 2.7 is to select x_i to be the least variable according to $\prec$ in E on line 2. Also x_j is selected to be the least variable in S according to $\prec$ on line 10. The original tech report [47] gives a proof of termination. This style of using a variable ordering to ensure termination is generally known as *Bland's rule*, and is a common way of constructing terminating variants of simplex algorithms [19, 56, 98].

[5] This explanation is non-minimal as not all of $A\,\mathcal{X} = 0$ may be needed. Minimal theory valid conflicts are discussed in Section 2.2.9.

2.2 Refining the Abstract Algorithm

The previous section gave an abstract view of the SIMPLEXFORSMT algorithm. This section refines and develops the concepts in 2.1 into what is needed for a high quality implementation of SIMPLEXFORSMT. We will weaken previous assumptions, elaborate data structures and flesh out additional features. Small original contributions are discussed and will be labeled as such in the relevant subsections.

2.2.1 Delta Arithmetic

A major advantage of the problem formalization given in Sec. 2.1 is that it can be extended to allow for handling strict and non-strict inequalities in a uniform manner without increasing the number of rows or variables. Inequalities (both strict and non-strict) are conceptually expanded to include an implicit, positive, infinitesimal, global variable δ. The strict inequality $x < d$ is satisfied iff there exists some $\delta > 0$ such that $x + \delta \leqslant d$ or $x \leqslant d - \delta$. Similarly, $x > d$ iff $x \geqslant d + \delta$ for some $\delta > 0$. The right-hand-side of inequalities can be represented as a non-strict inequality between x and $d + e \cdot \delta$ where d and e are real numbers. As δ is implicit, this is stored as a pair of real numbers $\langle d, e \rangle$. The following transformations suffices to handle all of the *admissible* candidate right-hand sides:

$$\begin{array}{lclcl}
x \leqslant d & \mapsto & x \leqslant d + 0 \cdot \delta & \mapsto & x \leqslant \langle d, 0 \rangle \\
x < d & \mapsto & x \leqslant d + f \cdot \delta & \mapsto & x \leqslant \langle d, f \rangle \\
x \geqslant d & \mapsto & x \geqslant d + 0 \cdot \delta & \mapsto & x \geqslant \langle d, 0 \rangle \\
x > d & \mapsto & x \geqslant d + g \cdot \delta & \mapsto & x \geqslant \langle d, g \rangle \\
x = d & \mapsto & x = d + 0 \cdot \delta & \mapsto & x = \langle d, 0 \rangle \\
x \neq d & \mapsto & x \neq d + 0 \cdot \delta & \mapsto & x \neq \langle d, 0 \rangle
\end{array} \tag{2.7}$$

where $f < 0$, $g > 0$, and e is either 0, f or g. In the implementation, the values for f and g are -1 and 1 during this transformation (i.e. $e \in \{-1, 0, 1\}$). However, the proofs focus on the sign of e instead of its value.

Given a subfield F of $\mathbb{R}$, we define the *delta extension* of F as the set of pairs $\langle d, e \rangle \in F^2$ interpreted as $d + e\delta$ and denote the set of values as δF. The delta extension of F is a finite dimension vector space over F. The δF values may be added or subtracted from each other and scaled by F values. For generality, the proofs in this subsection will be over $\delta\mathbb{R}$ while in the implementation, the only delta extension will be $\delta\mathbb{Q}$ (delta-rationals).

2.2.1.1 Delta-Extension Comparisons

The implicit variable δ is intended to act as an infinitesimal. Specifically, it can be set to an arbitrarily small positive $\mathbb{R}$ value. This requires care in defining a comparison operation. For example, we expect the following chain to hold: $4 < 4 + \delta < 5 - \delta < 5$. More formally, we define $<_\delta$ on δF such that for all sufficiently small real numbers the ordering is consistent with the $<$ order on the reals:

$$d_1 + e_1 \cdot \delta <_\delta d_2 + e_2 \cdot \delta \quad \equiv \quad (\exists \alpha > 0.\ \forall \beta \in (0, \alpha).\ d_1 + e_1 \cdot \beta < d_2 + e_2 \cdot \beta)\,.$$

This order is equivalent to using lexicographic ordering $<_{lex}$ over pairs of F^2. Before proving this, note that the admissible ordering $4 < 4 + \delta < 5 - \delta < 5$ example matches

$$\langle 4, 0\rangle <_{lex} \langle 4, 1\rangle <_{lex} \langle 5, -1\rangle <_{lex} \langle 5, 0\rangle\,.$$

Lemma 2.2. *If* $\langle d_1, e_1\rangle <_{lex} \langle d_2, e_2\rangle$, *then* $d_1 + e_1 \cdot \delta <_\delta d_2 + e_2 \cdot \delta$.

Proof. Suppose $\langle d_1, e_1\rangle <_{lex} \langle d_1, e_2\rangle$. Then either $d_1 < d_2$ or $d_1 = d_2 \wedge e_1 < e_2$.

- If $d_1 = d_2$ and $e_1 < e_2$, then $d_1 + e_1 \cdot \beta < d_2 + e_2 \cdot \beta$ holds for $\beta > 0$.
- Suppose $d_1 < d_2$ holds. Either $e_1 > e_2$ or $e_1 \leqslant e_2$ must hold.
  - If $e_1 \leqslant e_2$, then $d_1 + e_1 \cdot \beta < d_2 + e_2 \cdot \beta$ holds for all $\beta > 0$.
  - Suppose $e_1 > e_2$. Let $\alpha = \frac{d_2 - d_1}{e_1 - e_2}$. As $d_2 > d_1$ and $e_1 > e_2$, $\alpha > 0$. Let β be any value in the open interval $(0, \alpha)$.

$$0 < \beta < \frac{d_2 - d_1}{e_1 - e_2}$$

    Thus $\beta\,(e_1 - e_2) < d_2 - d_1$, and we can see by rewriting that $d_1 + e_1 \cdot \beta < d_2 + e_2 \cdot \beta$ holds.

Then for all of the cases above, there is some value of α (either $\alpha = \frac{d_2 - d_1}{e_1 - e_2}$ or $\alpha = 1$), such that $d_1 + e_1 \cdot \delta <_\delta d_2 + e_2 \cdot \delta$ holds. □

Lemma 2.3. *If* $d_1 + e_1 \cdot \delta <_\delta d_2 + e_2 \cdot \delta$, *then* $\langle d_1, e_1\rangle <_{lex} \langle d_2, e_2\rangle$.

Proof. Let α be a positive real value such that $d_1 + e_1 \cdot \beta < d_2 + e_2 \cdot \beta$ for all $\beta \in (0, \alpha)$. We first rewrite this to $d_1 - d_2 < (e_2 - e_1)\beta$.

- Suppose $e_1 \geqslant e_2$. As $\beta > 0$, $0 \geqslant (e_2 - e_1)\beta > d_1 - d_2$. Thus $d_1 < d_2$.

- Suppose $e_1 < e_2$. Let ξ be any value in $(0, (e_2 - e_1)\,\alpha)$. Then $\beta = \frac{\xi}{e_2 - e_1}$ is in the range $(0, \alpha)$, and $d_1 - d_2 < \xi$ must hold. As this holds for all candidate values of ξ,

  $$d_1 - d_2 \leqslant \inf\{\xi \in \mathbb{R} | 0 < \xi < (e_2 - e_1)\,\alpha\} = 0$$

  Hence, $d_1 \leqslant d_2$ and $e_1 < e_2$.

So in both cases $\langle d_1, e_1\rangle <_{lex} \langle d_2, e_2\rangle$. □

Corollary 2.4. $d_1 + e_1 \cdot \delta <_\delta d_2 + e_2 \cdot \delta$ *iff* $\langle d_1, e_2\rangle <_{lex} \langle d_2, e_2\rangle$

By the properties of lexicographic orders over total orders, comparison over δF, $<_\delta$, must be total. The order then extends to the relations $=_\delta$, $\leqslant_\delta$, $>_\delta$ and $\geqslant_\delta$. For brevity, we drop the subscript $_\delta$ in comparisons when distinguishing between $\mathbb{R}$ and $\delta\mathbb{R}$ is irrelevant to the discussion.

δF can then be treated as a vector space over F with a total ordering between elements. Using this insight, the key datastructures a, l and u are all internally δ extensions with the natural extensions to $\pm\infty$. Let $\mathbb{R}^+$ denote the set of non-negative real numbers, and $\mathbb{R}^-$ denote the set of non-positive real numbers.

$$\begin{aligned} a &: \mathcal{X} \mapsto \mathbb{R} \times \mathbb{R} \\ l &: \mathcal{X} \mapsto (\mathbb{R} \times \mathbb{R}^+) \cup \{-\infty\} \\ u &: \mathcal{X} \mapsto (\mathbb{R} \times \mathbb{R}^-) \cup \{+\infty\} \end{aligned} \tag{2.8}$$

All of the algorithms in Section 2.1 use a, l and u in ways that are compatible with vector spaces over $\mathbb{Q}$, and the main properties, invariants, and termination properties remain fundamentally unchanged.

$$T\mathcal{X} =_\delta 0,\ Ta =_\delta 0,\ l \leqslant_\delta a \leqslant_\delta u,\ \text{ etc.}$$

Thus $\delta\mathbb{R}$ allows for the algorithm to handle strict and non-strict inequalities uniformly. Switching to $\delta\mathbb{R}$ assignments introduces a slight discrepancy in that a is no longer a $\mathcal{T}_\mathbb{R}$ interpretation. Subsection 2.2.1.4 describes computing a value for α which subsection 2.2.11 will use to select a value for β such that substituting β for δ everywhere in the assignment a with this value yields a satisfying $\mathcal{T}_\mathbb{R}$ interpretation M.

2.2.1.2 Satisfaction in Delta Extensions

Next, we describe the relationship between the standard notion of real satisfaction and satisfaction of delta-extended comparisons over linear terms by delta-extended assignments.

Let p be an atom of the form $t \bowtie d$ with $\bowtie \in \{\leqslant, <, =, \neq, >, \geqslant\}$ where t is a linear term and d is a rational constant. The literal can be mapped to an admissible delta-extended atom of the form:

$$q \equiv t \bowtie_\delta d + e\delta = \begin{cases} t \geqslant_\delta d + e\delta & \bowtie \text{ is } > \text{ and } e > 0 \\ t \leqslant_\delta d + e\delta & \bowtie \text{ is } < \text{ and } e < 0 \\ t \bowtie_\delta d + 0\delta & \text{otherwise} \end{cases}$$

The conversion from p to some q formalizes the delta-arithmetic encoding of a single literal. A delta-extended assignment a_δ maps every real variable to a pair of real values $\langle d, e\rangle \in \mathbb{R}^2$ which represents $d + e\delta$. The extended assignment a_δ is extended to evaluating arbitrary linear terms in the following fashion:

- A variable x is evaluated as $\mathrm{Eval}(a_\delta, x) = a_\delta(x)$.
- A rational constant c is evaluated as $\mathrm{Eval}(a_\delta, c) = \langle c, 0\rangle$.
- The sum $s + t$ is evaluated as $\mathrm{Eval}(a_\delta, s + t) = \mathrm{Eval}(a_\delta, s) + \mathrm{Eval}(a_\delta, t)$.
- The multiplication by constants $c \cdot t$ is evaluated as $\mathrm{Eval}(a_\delta, c \cdot t) = \langle cd, ce\rangle$ where $\langle cd, ce\rangle = \mathrm{Eval}(a_\delta, t)$.

The special case to capture multiplication by a constant $c \cdot t$ corresponds to treating $\delta\mathbb{R}$ as a vector space. An extended assignment a_δ satisfies q whenever the delta extended comparisons hold over the pairs of real numbers. This is denoted $a_\delta \models_{\delta\mathbb{R}} q$ [with considerable abuse of notation]. We extend this notation of satisfaction to finite conjunctions of such q atoms and entailment $\bigwedge q_i \models_{\delta\mathbb{R}} q$ in the natural way.[6]

Let $M_{a_\delta,\beta}$ denote the real interpretation M generated by assigning to each real variable x_i the value $d + e\beta$ where $a_\delta(x_i) = \langle d, e\rangle$, i.e. replace the variable with the extended assignment and replace δ with the real constant β. Let q be a delta-encoding of a literal p with the right-hand side $\langle d_q, e_q\rangle$.

Lemma 2.5. *If an extended assignment* $a_\delta \models_{\delta\mathbb{R}} q$*, then there exists an* $\alpha > 0$ *such that for all* $\beta \in (0, \alpha)$ *the following holds* $M_{a_\delta,\beta} \models_{\mathbb{R}} p$.

Proof. Suppose $a_\delta \models_{\delta\mathbb{R}} q$. The form of q is $t \bowtie_\delta d_q + e_q\delta$. Suppose that the value of the left-hand side t evaluates to $d_t + e_t\delta$ under a_δ in delta-arithmetic, so that $d_t + e_t\delta \bowtie_\delta d_q + e_q\delta$. As the trichotomy property holds for $<_\delta$. the constants $d_t + e_t\delta$ and $d_q + e_q\delta$ are related by either $<_\delta$, $=_\delta$ or $>_\delta$.

- If $d_t + e_t\delta >_\delta d_q + e_q\delta$ holds, then either $d_t > d_q$ or $d_t = d_q$ and $e_t > e_q$. The symbol $\bowtie_\delta$ is either $\geqslant_\delta$ or $\neq_\delta$, and the relational symbol in p ($\bowtie$) is either $\geqslant$, $>$, or $\neq$. As $\bowtie$ is not $<$, $e_q \geqslant 0$.

[6] This is not generalized further as it is not closed under negation.

  - Suppose $e_t < e_q$. Both $e_q - e_t > 0$ and $d_t - d_q > 0$ hold. Let $\alpha = \frac{d_t - d_q}{e_q - e_t} > 0$. Then, for all $\beta \in (0, \alpha)$, $d_t - d_q > (e_q - e_t)\beta$. So, $d_t + e_t\beta > d_q + e_q\beta \geqslant d_q$, and $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.
  - See the appendix A.2.1 for the $e_t \geqslant e_q$ case.

- See the appendix A.2.1 for the $<_\delta$ and $=_\delta$ cases.

In all of these cases, $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ must hold for all $\beta \in (0, \alpha)$ for some $\alpha > 0$. □

Next we generalize Lemma 2.5 from literals to conjunctions of literals. Let the sequence of literals $p_1, \ldots, p_n$ generate a sequence of delta extended relations $q_1, \ldots, q_n$.

Lemma 2.6. *Let* q_i *be any admissible delta-encoding of the literal* p_i. *If* $a_\delta \models_{\delta\mathbb{R}} \bigwedge q_i$, *then there exists an* $\alpha > 0$ *such that for all* $\beta \in (0, \alpha)$ *that* $M_{a_\delta,\beta} \models_{\mathbb{R}} \bigwedge p_i$.

Proof. This follows directly from Lemma 2.5. Suppose $a_\delta \models_{\delta\mathbb{R}} \bigwedge q_i$. Then $a_\delta \models_{\delta\mathbb{R}} q_i$. So there exists some $\alpha_i > 0$ such that for all $\beta \in (0, \alpha_i)$, $M_{a_\delta,\beta} \models_{\mathbb{R}} p_i$ holds. Let $\alpha = \min \alpha_i$. Then for all $\beta \in (0, \alpha)$, $M_{a_\delta,\beta} \models_{\mathbb{R}} \bigwedge p_i$ holds. □

The naive converse of Lemma 2.5 would claim that if there exists an $\alpha > 0$ such that for all $\beta \in (0, \alpha)$ that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds, then $a_\delta \models_{\delta\mathbb{R}} q$. (Lemma 2.6 would be similar.) This claim does not hold due to a discontinuity introduced by selecting e. Suppose that p is $x + y > 5$ and q is the admissible encoding $x + y \geqslant \langle 5, 1\rangle$. Let a_δ be an extended assignment such that

$$a_\delta(x) = \left\langle 6, \frac{1}{2} \right\rangle \quad \text{and} \quad a_\delta(y) = \left\langle -1, -\frac{1}{4} \right\rangle.$$

The evaluation of $x + y$ is $\langle 5, \frac{1}{4}\rangle$. As $5 + \frac{1}{4}\beta > 5$ for all $\beta > 0$, $M_{a_\delta,\beta} \models p$. However, a_δ does not satisfy q as $\langle 5, \frac{1}{4}\rangle$ is less than $\langle 5, 1\rangle$. To fix this, we can instead show that there exists a satisfying extended assignment $a_{\zeta\delta}$ that multiplies the δ coefficients of the variables by a constant $\zeta \geqslant 1$. To fix the counter-example, suppose $\zeta = 4$, the assignment $a_{\zeta\delta}$ maps x to $\langle 6, 2\rangle$ and y to $\langle -1, -1\rangle$. This extended assignment satisfies q and the other criteria. The formalization and proofs for the converses of Lemmas 2.5 and 2.6 are in the appendix in section A.2.2.

Lemma 2.7. *Suppose* $M \models_{\mathbb{R}} p$. *Let* q *be a delta-encoding of the literal* p *and the assignment* a_δ *map all variables* x_i *in* p *to* $\langle x_i^M, 0\rangle$. *Then* $a_\delta \models_{\delta\mathbb{R}} q$.

Proof. See Appendix A.2.3 for a straightforward proof. □

Section 2.24 gives an algorithm to compute α directly from an assignment a_δ satisfying some $\bigwedge q_i$ to generate a class of satisfying interpretations $M_{a_\delta,\beta}$.

2.2.1.3 Delta-Extension Entailment

Let q be any delta-extended literal corresponding to a $\mathcal{T}_{\mathbb{R}}$ literal p. Specifically q has the form $t \bowtie_\delta d + e\delta$ where $\bowtie_\delta$ is either $\geqslant_\delta$, $\leqslant_\delta$, $=_\delta$, or $\neq_\delta$, $e > 0$ only if $\bowtie_\delta$ is $\geqslant_\delta$ and $e < 0$ only if $\bowtie_\delta$ is $\leqslant_\delta$.

Lemma 2.8. *If* $\bigwedge q_i \models_{\delta\mathbb{R}} q$*, then* $\bigwedge p_i \models_{\mathbb{R}} p$ *where* q_i *is an admissible delta-encoding of the literal* p_i*.*

Proof. Assume that $\bigwedge q_i \models_{\delta\mathbb{R}} q$. This means that for all assignments a_δ, if $a_\delta \models_{\delta\mathbb{R}} \bigwedge q_i$ holds, then $a_\delta \models_{\delta\mathbb{R}} q$ holds. Let M be any interpretation satisfying $\bigwedge p_i$. Let a_δ be an assignment that maps x_i to $\langle x_i^M, 0\rangle$. Then $a_\delta \models_{\delta\mathbb{R}} q_i$ for each q_i. Thus $a_\delta \models_{\delta\mathbb{R}} q$. Structurally, $q \equiv t \bowtie_\delta d + e\delta$. The evaluation of t by M is t^M. Thus the evaluation of t under a_δ is $\langle t^M, 0\rangle$. For all cases, we know that $t^M + 0\delta \bowtie_\delta d + e\delta$ holds.

- Suppose that $e < 0$. Then $\bowtie_\delta$ must be $\leqslant_\delta$ and p has the form $t < d$. Because q is satisfied by a_δ, we have $t^M + 0\delta \leqslant_\delta d + e\delta$. To satisfy this $t^M \leqslant d$ and $t^M \neq d$ as $e < 0$. Thus $t^M < d$ and $M \models_{\mathbb{R}} p$.
- See Appendix A.2.4 for a proof that includes the $e = 0$ and $e < 0$ cases.

□

(The reverse direction of the previous lemma appears in Appendix A.2.4.)

The core entailment rules for polynomials over the reals continue to apply to the δ extension. We claim that the following entailment rules hold [without proof] for polynomials s, t, u, and v:

$$\begin{aligned} t =_\delta s &\models_{\delta\mathbb{R}} \alpha t =_\delta \alpha s \\ (d \bowtie_\delta e) \wedge (t =_\delta s) &\models_{\delta\mathbb{R}} t + d \bowtie_\delta s + e \\ t \geqslant_\delta s &\models_{\delta\mathbb{R}} \alpha t \geqslant_\delta \alpha s \text{ for } \alpha \in \mathbb{R}^+ \\ t \bowtie_\delta s &\models_{\delta\mathbb{R}} t + d \bowtie_\delta s + d \end{aligned} \tag{2.9}$$

2.2.1.4 Order-Preserving Ranges

An issue with using δ-extended arithmetic is that the assignment a_δ is not directly a $\mathcal{T}_{\mathbb{R}}$ interpretation. When we later construct models, this will involve selecting a value β to get $M_{a_\delta,\beta}$. Suppose that the theory satisfies the assertion $x \neq y$ by the assignment $a_x = 4 + \delta$, and $a_y = 5 - \delta$. If the value of β is selected to be $\frac{1}{2}$, then $M_{a_\delta,\beta}$ does not satisfy $x \neq y$. The values of β we want are those such that no pair of $\delta\mathbb{R}$ values $d + e\delta$ and $d' + e'\delta$ involved in constructing the satisfying model that are disequal $(d + e\delta \neq_\delta d' + e'\delta)$ become equal once δ is replaced by the value β. This becomes important once the procedure is extended to handle disequalities in Section 2.2.10.

A range $(0, \alpha)$ is *order-preserving* over a set Q of $\delta\mathbb{R}$ if for all $\beta \in (0, \alpha)$ if $d + e\delta$ and $\mu + \nu\delta$ are in Q and $d + e\delta <_\delta \mu + \nu\delta$, then $d + e\beta < \mu + \nu\beta$. Intuitively, the order between the two is preserved by β.

We give an algorithm to compute α such that $(0, \alpha)$ is order-preserving over a finite set Q. (This is a minor contribution to the existing literature.) The algorithm begins by sorting Q in increasing order according to $<_{\delta Q}$ to get an enumeration $d_1 + e_1\delta, \dots, d_K + e_K\delta$ with $K = |Q|$. We append an extra element $d_{K+1} + e_{K+1}\delta$ with $d_{K+1} = d_K + 1$ and $e_{K+1} = e_K - 1$.[7]

$$d_1 + e_1\beta <_\delta d_2 + e_2\beta <_\delta \cdots <_\delta d_K + e_K\beta <_\delta d_{K+1} + e_{K+1}\beta \tag{2.10}$$

We take the differences between $(d_{i+1} + e_{i+1}\delta) - (d_i + e_i\delta)$ to get a series of K values $d'_1 + e'_1\delta, \dots, d'_K + e'_K\delta$. All of these differences are positive, $d'_i + e'_i\delta >_\delta 0 + 0\delta$. We will now show that we only have to examine the pairs where $d'_i > 0$ and $e'_i < 0$. For each value either $d'_i > 0$ or $d'_i = 0$ and $e'_i > 0$.

- Suppose $d'_i = 0$ and $e'_i > 0$. As $d_{i+1} + e_{i+1}\delta >_\delta d_i + e_i\delta$, we must have $d_{i+1} = d_i$ and $e_{i+1} > e_i$. Then for all $\beta > 0$, the inequality holds after replacement.
- Suppose $d'_i > 0$ and $e'_i \geqslant 0$. Thus $e_{i+1} \geqslant e_i$. So for all $\beta > 0$, the inequality $d_{i+1} + e_{i+1}\beta \geqslant d_i + e_i\beta$ holds.

Let $\Upsilon = \left\{ -\frac{d'_i}{e'_i} \middle| d'_i > 0, e'_i < 0 \right\}$. Note that each value $-\frac{d'_i}{e'_i}$ in Υ must be strictly greater than 0. We now select a value of α to be the minimum value in Υ.

Lemma 2.9. *The range* $(0, \alpha)$ *is order-preserving over* Q.

Proof. The set Υ is non-empty as $d'_K = d_{K+1} - d_K = 1$ and $e'_K = e_{K+1} - e_K = -1$. By our definition of α:

$$\alpha = \min -\frac{d'_i}{e'_i} \text{ for all } d'_i > 0 \text{ and } e'_i < 0.$$

Let $\beta \in (0, \alpha)$. We now show that $d_{i+1} + e_{i+1}\beta > d_i + e_i\beta$ for i ranging from 1 to K. As was shown during the discussion of the algorithm, if it is not the case that $d'_i > 0$ and $e'_i < 0$, then the order $d_{i+1} + e_{i+1}\delta >_\delta d_i + e_i\delta$ is preserved for all $\beta > 0$. The interesting case is when $d'_i > 0$ and $e'_i < 0$, or $d_{i+1} > d_i$ and $e_{i+1} < e_i$. Then $-\frac{d'_i}{e'_i} \geqslant \alpha > \beta > 0$. We know that $-e_i$ is positive so multiplying

[7] Adding the element $d_{K+1} + e_{K+1}$ is simply to reduce the number of case splits in the construction. Having it covers the $K = 0$ and $K = 1$ cases and ensures the set Υ is non-empty.

both sides of $-\frac{d'_i}{e'_i} > \beta$ by $-e_i$ yields $d'_i > -e'_i\beta$. Replacing d'_i and e'_i by the differences $d_{i+1} - d_i$ and $e_{i+1} - e_i$ and rewriting the inequality gives the desired claim for all adjacent pairs, $d_i + e_i\beta < d_{i+1} + e_{i+1}\beta$. We have now shown that the order of all adjacent pairs in Q are preserved by the range $(0, \alpha)$. Then using (2.10), we can conclude that:

$$d_1 + e_1\beta < d_2 + e_2\beta < \cdots < d_K + e_K\beta < d_{K+1} + e_{K+1}\beta.$$

□

2.2.2 Inference by Farkas' Lemma

Inference in Simplex solvers is traditionally formalized in terms of some variant of Farkas' lemma [50]. A traditional variant of Farkas' lemma equates the existence of a satisfying solution with the non-existence of a negative constant being the result of a positive sum of inequalities on rows.

Lemma 2.10 (Farkas' lemma)**.** *Let* B *be a matrix and let* b *be a vector. Then the system of linear inequalities* $Bx \leqslant b$ *has a solution for* x *if and only if for every vector* y *such that* $y \geqslant 0$ *and* $y^T B = 0$, $y^T b \geqslant 0$.

Proof. See [98, Section 7.3] for a discussion of the proof. □

This section gives a variant of Farkas' lemma that suffices for all of the usages of this theorem in this thesis. This variant explicitly bridges the gap between symbolic logical formulas and the numeric properties of these formulas. It also follows the formulation of the constraints into the tableau and bounds on variables. And it allows for strict inequalities via delta-encodings, variables without bounds, and having both lower and upper bounds present.

Intuitively, we are going to take any linear combination of the input rows $y^T (A\mathcal{X} = 0)$ to derive an entailed equality $z \cdot \mathcal{X} = 0$ where $z^T = y^T A$. A subset of the variables that appear in the row ($z_k \neq 0$) are selectively relaxed to either their upper or lower bounds to derive an entailed inequality on the remaining variables.

This section uses extended arithmetic where $\pm\infty$ are treated as constants. The constant $+\infty$ is treated as short hand for there does not exist a finite bound. As such $c \cdot +\infty = +\infty$ for $c > 0$, and $c \cdot +\infty = -\infty$ for $c < 0$, $c - \infty = -\infty$, etc. (We are careful to ensure that $\infty - \infty$ and $0 \cdot \pm\infty$ are never encountered.)

Let $A\mathcal{X} = 0$ be a system of n linear equalities over the variables $\mathcal{X}$. The upper and lower bounds for variables u and l are treated as vectors over $(\mathbb{R} \times \mathbb{R}^- \cup \{+\infty\})^n$ and $(\mathbb{R} \times \mathbb{R}^+ \cup \{-\infty\})^n$ respectively. The vectors $\mathfrak{L}$ and $\mathfrak{U}$ are n-dimensional vectors of `QF_LRA` literals that point-wise rewrite using (2.7) over

$\mathcal{X}$ into the constraints $l \leqslant \mathcal{X} \leqslant u$. When l_i is finite, $l_i \leqslant_\delta x_i$ is an admissible encoding of $\mathfrak{L}_i$ and similarly, $x_i \leqslant_\delta u_i$ is an admissible encoding of $\mathfrak{U}_i$. When $l_i = -\infty$, then $\mathfrak{L}_i =$ **true** i.e. there is no lower bound for x_i. (Similarly, $u_i = +\infty$ iff $\mathfrak{U}_i =$ **true**.) The vectors $\mathfrak{L}$ and $\mathfrak{U}$ are designed to correspond to the assertions to the theory solver entailing the strongest upper and lower bounds on an individual variable. For example, if the tightest asserted bounds on x and y are $x \leqslant 5$ and $y > 2$ (and $y > 2$ is selected to be encoded as $y \geqslant 2 + \delta$), then the vectors $\mathfrak{L}_i$, l_i, $\mathfrak{U}_i$ and u_i are:

	$\mathfrak{L}_i$	l_i	$\mathfrak{U}_i$	u_i
x	**true**	$-\infty$	$x \leqslant 5$	$\langle 5, 0\rangle$
y	$y > 2$	$\langle 2, 1\rangle$	**true**	$+\infty$

We now formalize the various components of deriving the inequalities. Let y be any n-dimensional row vector over $\mathbb{R}$. Let $z = yA$ i.e. z is the result of a linear combination of the rows of A. The constraint $z \cdot \mathcal{X} = 0$ is entailed by a subset of the constraints in $A\mathcal{X} = 0$. The minimal set of equalities that participate in the entailment is denoted $\mathfrak{R}_\mathbb{R}$.

$$\mathfrak{R}_\mathbb{R} = \{A_k \cdot \mathcal{X} = 0 \mid y_k \neq 0, A_k \neq 0\}$$
$$\mathfrak{R}_\delta = \{A_k \cdot \mathcal{X} =_\delta 0 \mid y_k \neq 0, A_k \neq 0\}$$

The set $\mathfrak{R}_\delta$ is the same set of equalities, but over delta-extended arithmetic.

The subset of indices of variables to be replaced by their lower bounds will be denoted $\mathcal{L}$, and those replaced by their upper bounds will be denoted $\mathcal{U}$. We require that $\mathcal{L} \subseteq \{i|z_i > 0\}$ and $\mathcal{U} \subseteq \left\{j|z_j < 0\right\}$. The set of indices that are non-zero but will not be replaced by a bound are denoted $\mathcal{F}$.

$$\mathcal{F} = \{k|z_k \neq 0\} \setminus (\mathcal{L} \cup \mathcal{U})$$

Let $\mathcal{A}'_\delta$ be the set of delta constraints actively involved in $\mathfrak{R}_\delta$, $\mathcal{L}$ and $\mathcal{U}$:

$$\mathcal{A}'_\delta = \mathfrak{R}_\delta \cup \{x_i \geqslant_\delta l_i | i \in \mathcal{L}\} \cup \left\{x_j \leqslant_\delta u_j | j \in \mathcal{U}\right\}$$

and $\mathcal{A}'_\mathbb{R}$ be the real analog of these assertions

$$\mathcal{A}'_\mathbb{R} = \mathfrak{R}_\mathbb{R} \cup \{\mathfrak{L}_i | i \in \mathcal{L}\} \cup \left\{\mathfrak{U}_j | j \in \mathcal{U}\right\}.$$

Let $\gamma = \sum_{i \in \mathcal{L}} z_i l_i + \sum_{j \in \mathcal{U}} z_j u_j$. We now show that: $-\sum_{k \in \mathcal{F}} z_k x_k \geqslant_\delta \gamma$.

Lemma 2.11. *In extended arithmetic, either* γ *is* $-\infty$*, or* γ *is* $d + e\delta$ *with* $e \geqslant 0$ *and for all* $i \in \mathcal{L}$ *and all* $j \in \mathcal{U}$*,* l_i *and* u_i *are finite.*

Proof. By definition, γ is $\sum_{i \in \mathcal{L}} z_i l_i + \sum_{j \in \mathcal{U}} z_j u_j$ with all $z_i > 0$ and $z_j < 0$. If there are any non-finite terms l_i or u_j, $z_i l_i = -\infty$ and $z_j u_j = -\infty$ and the inner sum is $-\infty$ in extended arithmetic.

If all of the terms l_i or u_j are finite, then each $l_i = d_i + e_i\delta$ with $e_i \geqslant 0$ and $u_j = d'_j + e'_j\delta$ with $e'_j \leqslant 0$. Each $z_i e_i \geqslant 0$ and $z_j e'_j \geqslant 0$, thus $\sum_{i\in\mathcal{L}} z_i e_i + \sum_{j\in\mathcal{U}} z_j e'_j \geqslant 0$. Finally, γ must then be of the form

$$\gamma = \left(\sum_{i\in\mathcal{L}} z_i d_i + \sum_{j\in\mathcal{U}} z_j d'_j\right) + \left(\sum_{i\in\mathcal{L}} z_i e_i + \sum_{j\in\mathcal{U}} z_j e'_j\right)\delta.$$

□

The following lemma generalizes all of inference that the theory solver does. Before showing the lemma, it is worth noting that the author has chosen to always frame inference in terms of minimization. This combined with other conventions create a number of extra negation symbols. Hence, $-\sum_{k\in\mathcal{F}} z_k x_k \geqslant_\delta \gamma$ appears instead of $\sum_{k\in\mathcal{F}} z_k x_k \leqslant_\delta -\gamma$ in the following lemma.

Lemma 2.12. *$\mathcal{A}'_\delta \models_{\delta\mathbb{R}} -\sum_{k\in\mathcal{F}} z_k x_k \geqslant_\delta \gamma$ if γ is finite.*

Proof. Suppose γ is $d + e\delta$ in extended delta arithmetic. As γ is finite, for all $i \in \mathcal{L}$, L_i has the form $d_i + e_i\delta$ with $e_i \geqslant 0$, and for all $j \in \mathcal{U}$, U_j has the form $d'_j + e'_j\delta$ with $e'_j \leqslant 0$. Let a_δ be any delta assignment of $\mathcal{A}'_\delta$.

$$a_\delta \models_{\delta\mathbb{R}} \bigwedge_{y_k\neq 0} (A_k \cdot \mathcal{X} =_\delta 0) \wedge \bigwedge_{i\in\mathcal{L}} (x_i \geqslant_\delta d_i + e_i\delta) \wedge \bigwedge_{j\in\mathcal{U}} \left(x_j \leqslant_\delta d'_j + e'_j\delta\right) \qquad (2.11)$$

Each individual constraint ($A_k \cdot \mathcal{X} =_\delta 0$, $x_i \geqslant_\delta d_i + e_i\delta$ and $x_j \leqslant_\delta d'_j + e'_j\delta$) holds in a_δ. This interpretation then also satisfies any linear combination of the $A_k \cdot \mathcal{X} =_\delta 0$ constraints, in particular,

$$a_\delta \models_{\delta\mathbb{R}} yA \cdot \mathcal{X} =_\delta 0 \quad \text{and} \quad -z \cdot \mathcal{X} =_\delta 0.$$

This symbolic equality can be broken into

$$\sum_{i\in\mathcal{L}} -z_i x_i + \sum_{j\in\mathcal{U}} -z_j x_j + \sum_{k\in\mathcal{F}} -z_k x_k =_\delta 0$$

The inequalities $x_i \geqslant_\delta d_i + e_i\delta$ for all $i \in \mathcal{L}$ entail that $a_\delta \models_{\delta\mathbb{R}} z_i x_i \geqslant_\delta z_i d_i + z_i e_i\delta$ as $z_i > 0$, or $z_i x_i - z_i d_i - z_i e_i\delta \geqslant_\delta 0 + 0\delta$. Similarly, for each $j \in \mathcal{U}$, $a_\delta \models_{\delta\mathbb{R}} z_j x_j - z_j d'_j - z_j e'_j\delta \geqslant_\delta 0 + 0\delta$. Then a_δ must also satisfy the inequality

$$\sum_{i\in\mathcal{L}} z_i(-x_i + x_i - d_i - e_i\delta) + \sum_{j\in\mathcal{U}} z_j(-x_j + x_j - d'_j - e'_j\delta) - \sum_{k\in\mathcal{F}} z_k x_k \geqslant_\delta 0 + 0\delta$$

After combining like terms this becomes,

$$a_\delta \models_{\delta\mathbb{R}} -\sum_{k\in\mathcal{F}} z_k x_k \geqslant_\delta \sum_{i\in\mathcal{L}} (z_i d_i + z_i e_i \delta) + \sum_{j\in\mathcal{U}} \left(z_j d_j' + z_j e_j' \delta\right)$$

By the definition of γ,

$$\gamma = d + e\delta = \left(\sum_{i\in\mathcal{L}} z_i d_i + \sum_{j\in\mathcal{U}} z_j d_j'\right) + \left(\sum_{i\in\mathcal{L}} z_i e_i + \sum_{j\in\mathcal{U}} z_j e_j'\right)\delta.$$

□

The δ arithmetic internal to the theory solver can be efficiently transformed back into proofs over $\mathcal{T}_{\mathbb{R}}$ terms to communicate with the THEORYENGINE.

Corollary 2.13. *Suppose* $\gamma = d + e\delta$. *Either*

- $e = 0$ *and* $\mathcal{A}_{\mathbb{R}}' \models_{\mathbb{R}} -\sum_{k\in\mathcal{F}} z_k x_k \geqslant d$, *or*
- $e > 0$ *and* $\mathcal{A}_{\mathbb{R}}' \models_{\mathbb{R}} -\sum_{k\in\mathcal{F}} z_k x_k > d$.

Proof. This immediately falls out as a combination of Lemmas 2.12 and 2.8. □

The following corollary connects the lemma back to the satisfiability of the input constraints. Intuitively, if replacing all of the variables by their bounds yields a γ such that $0 > \gamma$ and $\gamma > 0$, then a contradiction has been derived and the input assertions are unsatisfiable.

Corollary 2.14. *If* $\mathcal{F} = \varnothing$ *and* $\gamma = d + e\delta >_\delta 0 + 0\delta$, *then* $\mathcal{A}_{\mathbb{R}}' \models_{\mathbb{R}}$ **false**.

Proof. If $\mathcal{F}$ is empty, then $\mathcal{L}$ is equal to $\{i | z_i > 0\}$ and $\mathcal{U}$ is equal to $\left\{j | z_j < 0\right\}$ (rather than strict subsets). In this case, all of the variables on the left-hand side of Lemma 2.12 are being canceled so the resulting left-hand side is $0 + 0\delta$. Thus $\mathcal{A}_{\mathbb{R}}' \models_{\mathbb{R}} 0 + 0\delta \geqslant_\delta \gamma$ which is inconsistent as $d + e\delta >_\delta 0 + 0\delta$. □

The vector y, the constraints that form A, and $\mathcal{L}$ and $\mathcal{U}$ can be converted back into the input to the traditional Farkas' lemma. With some conceptual abuse, the sums of inequalities Farkas' lemma is performing

$$\sum y_j \left(\sum c_j x_j \geqslant d_j\right) \text{ with } y_j \geqslant 0$$

provide a simple uniform format for witnessing the correctness of the derived entailment to external proof checking tools, e.g. LFSC [91]. The main uses of such witnesses in SMT is the derivation of interpolants (which are outside of the scope of this thesis) [28]. See Section A.2.5 in the Appendix for more on the construction of such witnesses.

It should now be sufficiently clear that for QF_LRA there is no conceptual cost in implicitly converting between δ arithmetic and real arithmetic for the purposes of deducing entailed bounds. For brevity, we begin to write entailments without using δ-arithmetic. Such inequalities are never intended to leave the theory solver; however, it is clear that they inform what can leave the theory solver. ($x \geqslant_\delta d + \delta$ is treated as $x > d$ in external interactions.)

In following sections of this chapter, Lemma 2.12 is mostly applied when variables in $\mathcal{L}$ and $\mathcal{U}$ are assigned to be equal to their bounds. In this case, the sum of the remaining variables are minimized by their current assignment.

Corollary 2.15. *Let* a *be any* n*-dimensional vector of pairs of* $\mathbb{R}$ *such that* $Aa = 0$ *and for all* $i \in \mathcal{L}$ *and* $j \in \mathcal{U}$ *it is the case that* $a_i = l_i$ *and* $a_j = u_j$*. Then*

$$\mathcal{A}'_\delta \models_{\delta\mathbb{R}} -\sum_{k \in \mathcal{F}} z_k x_k \geqslant_\delta -\sum_{k \in \mathcal{F}} z_k a_k$$

Proof. As each $a_i = l_i$ and $a_j = l_j$ for $i \in \mathcal{L}$ and $j \in \mathcal{U}$, these are all finite. From the previous lemma, $\mathcal{A}'_\delta \models_{\delta\mathbb{R}} -\sum_{k \in \mathcal{F}} z_k x_k \geqslant_\delta \gamma$. By the definition of gamma,

$$\gamma = \sum_{i \in \mathcal{L}} z_i l_i + \sum_{j \in \mathcal{U}} z_j u_j = \sum_{i \in \mathcal{L}} z_i a_i + \sum_{j \in \mathcal{U}} z_j a_j$$

Then as $Aa = 0$ and $yAa = 0$,

$$\sum_{i \in \mathcal{L}} z_i a_i + \sum_{j \in \mathcal{U}} z_j a_j + \sum_{k \in \mathcal{F}} z_k a_k = 0$$

and $-\sum_{k \in \mathcal{F}} z_k a_k = \gamma$. □

2.2.3 Dynamically Adding Variables

We previously assumed a global preprocessing that transforms the input ϕ into $\phi'' \wedge A\,\mathcal{X} = 0$ (Section 2.2). This reduced ϕ'' to a formula containing only single variable inequalities under the assumption $A\,\mathcal{X} = 0$ held. This simplifies adding auxiliary variables and setting up the tableau T, but it is problematic as it does not allow for new atoms or variables to be added on-the-fly.

CVC4's arithmetic theory solver allows for variables to be added on demand. A unique *variable id* is assigned to nodes of linear sums over variables and is implemented as an unsigned machine word. Within the theory solver, variables are treated as their identifiers. This enables efficient array based implementations of maps using identifiers as keys. This also introduces a linguistic gap between the internals of the simplex decision procedure which knows about variable ids, but cannot communicate about assertions on the stack in order to interact with the THEORYENGINE. The theory solver bridges this gap, and is made responsible for coordinating with the outside world.[8]

To bridge this gap, Nodes for linear terms (see Sec. 1.5.10) will be assigned to variable ids. This is implemented using bidirectional maps from nodes to ids, NodeToId : $\texttt{Nodes} \mapsto \mathcal{X}$, and ids to nodes, orig : $\mathcal{X} \mapsto \texttt{Nodes}$. When a structural variable (a Node) x is seen in an assertion for the first time, x is given a variable id ι through a bi-directional pair of maps, $\text{NodeToId}(x) \leftarrow \iota$ and $\text{orig}(\iota) \leftarrow x$. The variable ι cannot yet have a relationship to any other variable so it is safe to give it an arbitrary initial assignment, $a(\iota) \leftarrow 0$, to assign it the trivial bounds $l(\iota) \leftarrow -\infty$ and $u(\iota) \leftarrow +\infty$, and to make the variable non-basic ($\iota \in \mathcal{N}$) and mark ι as structural.

For auxiliary variables, we will be interested in normalized linear terms t without constant offsets, $t \equiv \sum_{i=1}^{N} c_i x_i$ where $N \geqslant 2$ (otherwise the constraint is on a single structural variable), the variable nodes in the sum are strictly sorted, $x_1 \prec x_2 \prec \ldots \prec x_N$, and the coefficients ($c_i$) are non-zero. (The rewriter is used to ensure that these conditions are met.) When a node for a linear term t of this form is seen in an assertion for the first time, we check that all structural variables x_i have been setup as before, and t is given a variable id κ. ($\text{NodeToId}(t) \leftarrow \kappa$ and $\text{orig}(\kappa) \leftarrow t$.) Let ι_i be the variable id for the i'th structural variable, $\iota_i = \text{NodeToId}(x_i)$. The row T_κ is conceptually added at this point to the implicit matrix of original constraints A by extending the matrix with the row $-1 \cdot \kappa + \sum c_i \iota_i$. An initial assignment is computed for κ using t and the current assignments to x_i, $a(\kappa) \leftarrow \sum_{i=1}^{N} c_i a(\iota_i)$. Since some structural variables may have become basic since their introduction, the row added to T then removes these variables using their current rows,

$$-1 \cdot \kappa + \sum_{i=1}^{N} c_i \iota_i + \sum_{\iota_i \in \mathcal{B}} c_i T_i.$$

The variable κ can be given the trivial bounds $l_\kappa \leftarrow -\infty$ and $u_\kappa \leftarrow +\infty$. Finally, κ is marked as auxiliary and added to the set $\mathcal{B}$. Note this satisfies the main invariants (I1)-(I5).

The set $\mathcal{X}$ and the dimensions of A and T can easily be expanded on demand.[9] Removing variables is much more challenging because it is difficult to

[8] Linguistic gaps may appear initially unappealing, but these help promote good coding conventions for theory solvers with complex decision procedures. These require the theory solver to be an interface layer between the decision procedures and the SAT solver. This helps avoid bugs due to subtle re-entrant interactions between the SAT solver and the theory solver.

[9] As $\text{orig}(x_i) = x_i + A_i \mathcal{X}$ holds on any row i, either A or orig can be used to generate the

track when it is safe to do.

Auxiliary Variables are Internal Note that in this construction no actual variable (in the SMT sense) has been created. Nothing in the SMT solver except the theory solver has to be aware of the existence of κ (or x_κ). To enforce this, no Node for x_κ is ever created in CVC4, and the theory solver cannot use x_κ in conflicts or lemmas.

2.2.4 Constraints

The set $\mathcal{A}$ of assertions coming from the THEORYENGINE contains the conditions that the theory solver must check to see if they are satisfied. This is different from the set of literals that the theory has deduced that are entailed by $\mathcal{A}$. Suppose that the theory had the assertions $x - y \leqslant -5$ and $y - z \leqslant 5$. The following is then a valid deduction:

$$x - y \leqslant -5 \wedge y - z \leqslant 5 \models_{\mathbb{R}} x - z \leqslant 0 \tag{2.12}$$

The theory could send the clause $\neg(x - y \leqslant -5) \vee \neg(y - z \leqslant 5) \vee (x - z \leqslant 0)$ to the SAT solver as it is a valid theory lemma. However, if $x - z \leqslant 0$ is not in the original input, adding such lemmas may lead to non-termination of the SMT solver. Contrariwise, if something $x - z \leqslant 0$ entails has been preregistered as being in the original input, say $x - z \leqslant 1$, then propagating $x - z \leqslant 1$ (with the explanation $\{x - y \leqslant -5, y - z \leqslant 5\}$) to the SAT solver is likely to be beneficial.

To better support having internally known constraints and tracking proofs, it is useful to distinguish between assertions and constraints that are known to be entailed by the assertions. The arithmetic solver adds an intermediate layer called *constraints*. A constraint is fundamentally a variable id x (Section 2.2.3), a relation symbol $\bowtie \in \{=, \neq, \leqslant, \geqslant\}$, and a $\delta\mathbb{Q}$ value for the right-hand side v (Section 2.2.1). Each constraint that has been proven to be entailed has an attached *explanation*. The explanation is sat-context dependent, and it will only be set once per sat-context. We write $\mathfrak{e} \vdash c$ to denote that the constraint c has its proof index set to an explanation $\mathfrak{e}$, and $\nvdash c$ to denote that c has no explanation. An explanation $\mathfrak{e}$ is either a list of constraints that entail this constraint or is a flag that states that the constraint is equivalent to a constraint on the assertion trail.[10] If the explanation consists of just an assertion $\mathcal{A}_i$ in $\mathcal{A}$, the index of the assertion (i) is stored as well in a partial function a-index(c). The value of a-index(c) is set to i once the first p_i is asserted to the theory that is equivalent to c. A constraint

other.

[10] This is a rudimentary form of sequents over conjunctions.

c without an assertion index is denoted a-index$(c) = \bot$.

$$\frac{p_i \xrightarrow{\textsc{Rewriter}} t \bowtie_\delta v \qquad x_\kappa = \text{NodeToId}(t) \qquad \nvdash c : x_\kappa \bowtie_\delta v}{\mathfrak{e} : p_i \vdash c} \tag{2.13}$$

If the explanation is a list of constraints, these constraints must entail the node and must have their own explanations at the time of setting the explanation for c.

$$\frac{\{c_1, c_2, \dots, c_k\} \models_{\mathbb{R}} c \qquad \mathfrak{e}_1 \vdash c_1, \dots, \mathfrak{e}_k \vdash c_k \qquad \nvdash c}{\langle c_1, c_2, \dots, c_k \rangle \vdash c} \tag{2.14}$$

These explanations form a simple proof tree over $\vdash$ where all of the leaves are constraints that are rewritten assertions. (The $\nvdash c$ conditions above ensure that the explanation is only set once.) When a constraint c is propagated externally as a literal p, it can re-enter the theory solver as an assertion p_N. Thus a constraint c can have both an explanation in terms of a set of literals and at the same time c can have an associated assertion index, a-index$(c) \neq \bot$.

Only constraints that are associated with an assertion index are appropriate for conflicts. The constraints datastructure makes it simple to convert a set of constraints with any explanation ($\{\mathfrak{e} \vdash c\}$) into a set of constraints with set assertion indices. The following inference rule performs a restricted form of resolution that replaces a constraint c that is entailed by a sequence of constraints $\langle c_1, \dots, c_k \rangle$ by their explanations. (See section 1.3 for a description of resolution.)

$$\frac{c \wedge C \qquad \langle c_1, \dots, c_k \rangle \vdash c \qquad \text{a-index}(c) = \bot}{C \wedge \bigwedge \{c_1, c_2, \dots c_k\}} \ \textsc{Regress2Assertions} \tag{2.15}$$

Exhaustively applying (2.15) produces a set of constraints that can be directly converted into a set of assertions, $\{p_{\text{a-index}(c_i)}\}$. This calculation may implemented efficiently via depth-first-search.

If the constraint has been marked as having been preregistered, it is associated with it a single preregistration literal. For ease of implementation, whenever a constraint c is added so is its negation $\neg c$. The assertion index, the explanation and the preregistration literal are all sat-context dependent. As a matter of implementation the uniqueness of constraints can be enforced by a global constraint manager. This manager implements REGRESS2ASSERTIONS, backtracks explanations, maintains a hash map from nodes for literals to the corresponding constraint, and garbage collects constraints.

2.2.5 Handling Assertions

Assertions are fed into the arithmetic theory solver as a sat-context dependent stack, $\mathcal{A}$. The assertion p_i at position i on the stack is a literal of the form, $s \bowtie t$ where $\bowtie \in \{\neq, =, <, >, \leqslant, \geqslant\}$ and s and t are linear terms. If a constraint c is already in the constraint manager for the literal p_i, then p_i is already set up. (This is implemented via hashtable look ups.) Otherwise, p_i is associated with a constraint by the theory solver. The rewriter for the theory of arithmetic computes a sum-of-monomials normal form over terms. For a `QF_LRA` linear term t (without ites), abstractly this is

$$t \xrightarrow{\text{REWRITER}} c_0 + \sum_{i=1}^{N} c_i x_i$$

where: $N \geqslant 2$, the variable nodes in the sum are strictly sorted, $x_1 \prec x_2 \prec \ldots \prec x_N$, and the coefficients are non-zero $c_1, \ldots, c_N$ (though c_0 may be 0). To rewrite the literal $s \bowtie t$, first the term $s - t$ is rewritten to get a term of the form $c_0 + \sum_{i=1}^{N} c_i x_i$. The constant c_0 is then moved to the right-hand side to get $\sum_{i=1}^{N} c_i x_i \bowtie -c_0$. To ensure that all linearly independent sums are normalized to the same left-hand side, the relation is multiplied by $\frac{1}{c_1}$. (This potentially changes the direction of the inequality symbols $<$,$>$,$\leqslant$, and $\geqslant$.) The relation is then converted to delta rationals by changing the strict relation symbols $<, >$ into either $\leqslant_\delta$ or $\geqslant_\delta$ and the right-hand side $-\frac{c_0}{c_1}$ into a $\delta\mathbb{Q}$ using conversions in Equation 2.7. Let $d = -\frac{c_0}{c_1}$. The final result is a literal,

$$p_i \xrightarrow{\text{REWRITER}} \sum_{i=1}^{k} \frac{c_i}{c_1} x_i \bowtie'_\delta \langle d, e \rangle .$$

Returning to the example in 2.1, suppose the sequence of assertions was

$$\mathcal{A} = \left\langle -\frac{1}{5}x - y \geqslant 0,\ \ x = 5 + z,\ \ z - y \geqslant 1,\ \ y > -1 \right\rangle . \tag{2.16}$$

Assuming $x \prec y \prec z$, the sequence of rewritten constraints would be

$$x + 5y \leqslant_\delta \langle 0, 0\rangle ,\ \ x - z =_\delta \langle 5, 0\rangle ,\ \ y - z \leqslant_\delta \langle -1, 0\rangle ,\ \ y \geqslant_\delta \langle -1, 1\rangle . \tag{2.17}$$

After this transformation, two left-hand sides are linearly independent iff they are syntactically distinct. The new left-hand side t' is checked for having a variable id ι (and is set up if it does not have one). The constraint manager is then consulted for whether the constraint for $\iota \bowtie'_\delta d + e\delta$ already exists. If one does not exist, c is created. To avoid doing this process every time p_i comes in, the end-to-end result is then cached by the constraint manager to avoid future work: $\text{LitToConstraint}(s \bowtie t) \leftarrow c$. Processing the constraints in 2.17 introduces 3 new auxiliary terms s_1, s_2, and s_3 defined by:

$$s_1 = x + 5y, \;\; s_2 = x - z, \;\; s_3 = y - z$$

where these equalities are implicitly stored in the rows A_{s_1}, A_{s_2} and A_{s_3}, and the four constraints

$$c_1 : s_1 \leqslant \langle 0, 0 \rangle \,, \;\; c_2 : s_2 = \langle 5, 0 \rangle \,, \;\; c_3 : s_3 \leqslant \langle -1, 0 \rangle \,, \;\; c_4 : y \geqslant \langle -1, 1 \rangle \,.$$

We have so far assumed that no assertion is rewritten to the Boolean constants **true** or **false**. This can happen in examples such as

$$x + y = y + x \xrightarrow{\text{REWRITER}} \textbf{true}.$$

The literals that rewrite to **true** can be ignored while the literals that rewrite to **false** such as, $0 < 0$, are trivial conflicts.

Assertions are pulled off of the stack $\mathcal{A}$ in increasing order using GETASSERTIONSOFFSTACK in Figure 2.8. The position of the last processed assertion in the stack is saved in the variable `processedPos`. The variable `processedPos` is sat-context dependent and initially 0. This avoids having to bring in new assertions more than once within a sat-context. We additionally track whether or not all $\mathcal{A}_0$ through $\mathcal{A}_{\texttt{processedPos}}$ are known to be satisfied by the assignment α in the flag `rrstatus` (the real relaxation status). If there is any new assertion, the state of the real relaxation is conservatively set to unknown. The function ASSERTIONCASES dispatches a constraint c to either ASSERTLOWER, ASSERTUPPER, ASSERTEQUALITY, or ASSERTDISEQUALITY based on $\bowtie'$. (ASSERTEQUALITY and ASSERTDISEQUALITY are discussed in section 2.2.10.)

```
procedure GETASSERTIONSOFFSTACK
    while processedPos $< |\mathcal{A}|$ and no conflicts are on the output stream do
        rrstatus $\leftarrow$ Unknown
        $i \leftarrow$ processedPos
        processedPos $\leftarrow$ processedPos $+ 1$;
        if $p_i \xrightarrow{\text{REWRITER}}$ false then
            add (Conflict $p_i$) to the output stream
        else if $p_i \xrightarrow{\text{REWRITER}} \sum c_i x_i \bowtie_\delta v$ then
            $c \leftarrow$ TOCONSTRAINT($\sum c_i x_i \bowtie_\delta v$)
            if a-index(c) $= \bot$ then
                a-index(c) $\leftarrow i$
                if $\nvdash c$ then
                    $p_i \vdash c$
                    if $(\mathfrak{e} \vdash \neg c)$ then// $\neg c$ has an explanation $\mathfrak{e}$
                        output (Conflict REGRESS2ASSERTIONS($\{c, \neg c\}$))
        if no conflicts are on the output stream then
            ASSERTIONCASES (c)
```

Figure 2.8: Incrementally processing assertions.

2.2.6 Variable Status

Each variable is assigned a *status* describing the relationship between the variable and its bounds.

$$\text{Status}(x) = \begin{cases} \texttt{ABOVE\_UB} & a(x) > u(x) \geqslant l(x) \\ \texttt{AT\_UB} & u(x) = a(x) > l(x) \\ \texttt{BETWEEN} & u(x) > a(x) > l(x) \\ \texttt{FIXED} & u(x) = a(x) = l(x) \\ \texttt{AT\_LB} & u(x) > a(x) = l(x) \\ \texttt{BELOW\_LB} & u(x) \geqslant l(x) > a(x) \end{cases}$$

This value must be recomputed whenever either $a(x)$, $u(x)$, or $l(x)$ is updated. This includes modifying their values during an update operation, an ASSERT procedure, or backtracking the bounds. The cost of computing this status can be amortized into these operations and is a small constant overhead.[11] As there are only 6 possible states, these can be effectively implemented as an enumera-

[11] Backtracking bounds is the only operation that changes its complexity class. It goes from constant time to a linear-time operation. A straightforward amortized analysis can charge this operation against the worst-case complexity, $O(n)$, of the ASSERT procedure that set the bound.

tion attached to each variable. This provides an efficient mechanism for avoiding repeated exact precision inequality comparisons. Variable statuses can be grouped to capture additional properties such as the assignment of a variable equals its upper bound (2.18), the assignment of a variable is strictly less than its upper bound (2.19), or the variable being in the error set (2.20). It also provides an efficient mechanism for knowing when these properties change from one state to another.

$$\begin{aligned} a(x) = u(x) &\iff \text{Status}(x_k) \in \{\texttt{AT\_UB}, \texttt{FIXED}\} && (2.18)\\ a(x) < u(x) &\iff \text{Status}(x_k) \in \{\texttt{BETWEEN}, \texttt{AT\_LB}, \texttt{BELOW\_LB}\} && (2.19)\\ x \in E &\iff \text{Status}(x_k) \in \{\texttt{ABOVE\_UB}, \texttt{BELOW\_LB}\} && (2.20) \end{aligned}$$

2.2.7 Tableau

The tableau T is implemented using a sparse matrix representation. The sparse format only stores non-zero coefficients. All missing entries are implicitly zero. Each coefficient is a rational $T_{i,j}$ in a structure labeled with both an identifier for the row and the column variable. Each entry is a member of two doubly linked lists. The lists correspond to the non-zero entries in the row T_i and the column for the variable x_j, $(T^T)_j$. (Note that these lists are not sorted. The elements may appear in any order.) The entry has both forward and backward pointers for both column and row traversal. The head pointers into these lists are two arrays of size n: one for the beginnings of columns and one for the beginnings of rows. Iterating over rows and columns can be done in time proportional to the number of non-zero entries. We denote the number of non-zero entries of n-dimensional vector v as $\|v\|_s$. ($\|\cdot\|_s$ uses s to emphasize that it is a "size.") The following are constant time operations: maintaining the size (number of non-zero entries) of each row and column ($\|T_i\|_s$ and $\left\|(T^T)_j\right\|_s$), adding an entry $T_{i,j}$ if $T_{i,j}$ is known to be currently be 0, and setting an entry $T_{i,j}$ to 0 by unlinking it (given a pointer to the entry).

2.2.7.1 Row Addition

Implementing the tableau T in this fashion has the effect of making row addition efficient. Row addition adds to row k the row i scaled by a constant α, $T_k \leftarrow T_k + \alpha T_i$. This is the core matrix operation used. Row addition may be done in time $O(\|T_k\|_s + \|T_i\|_s)$ by using a temporary dense map. The algorithm is given in Fig. 2.9. One of the rows is loaded into the dense map and

```
1: procedure RowAddition($k, i, \alpha$, SC)
2:     tmp.clear() // tmp is a dense partial map with $O(n)$-entries
3:     tmp.add($\{j \mapsto T_{k,j} | T_{k,j} \neq 0\}$) // map each j to $T_{k,j}$ (via pointers) in tmp
4:     for all j s.t. $T_{i,j} \neq 0$ do
5:         $t \leftarrow$ if (j is a key in tmp) then tmp[j] else 0
6:         $t' \leftarrow t + \alpha T_{i,j}$
7:         if $\mathrm{sgn}(t') \neq \mathrm{sgn}(t)$ then
8:             SC $(k, j, \mathrm{sgn}(t), \mathrm{sgn}(t'))$
9:         if $t = 0$ then
10:            create entry for $T_{k,j} \leftarrow t'$ in linked lists $T_k$ and $(T^T)_j$
11:        else if $t' = 0$ then
12:            unlink entry $T_{k,j}$ from linked lists $T_k$ and $(T^T)_j$
13:        else
14:            update entry $T_{k,j} \leftarrow t'$
```

Figure 2.9: Compute $\alpha T_i + T_k$ and store the result in T_k. Reports the changes in sign to SC. $O(\|T_i\|_s + \|T_k\|_s)$

```
Require: $T_{i,j} \neq 0, i \neq j$
1: procedure Pivot-SC($i, j$, SC)
2:     RowAddition $\left(j, i, -\frac{1}{T_{i,j}}, \mathrm{SC}\right)$ // Create row j s.t. $T_{j,j} = -1$
3:     for all k such that $T_{k,j} \neq 0 \wedge k \neq j$ do // iterate over $(T^T)_j$
4:         RowAddition $(k, j, T_{k,j}, \mathrm{SC})$
5:     $(\mathcal{B}, \mathcal{N}) \leftarrow (\mathcal{B} \cup \{j\} \setminus \{i\}, \mathcal{N} \cup \{i\} \setminus \{j\})$
```

Figure 2.10: Pivot x_i and x_j and report sign changes to SC.

then row addition proceeds as normally.[12] In section 2.2.8, it will be helpful to know when an entry changes its sign. The procedure $\mathrm{SC}(i, j, \mathrm{sgn}(t), \mathrm{sgn}(t'))$ is a call back procedure for reporting that the value of the sign function on $\mathrm{sgn}(T_{i,j})$ changed from $\mathrm{sgn}(t)$ to $\mathrm{sgn}(t')$. For now, SC can be assumed to be a no-op. (See Fig. 2.11 for the actual implementation of SC.) Figure 2.10 adapts the pivot operation to also report sign changes on SC.

[12] A slightly more involved version of RowAddition can load the row being added from into the temporary map. While not asymptotically more efficient, then T_i only has to be loaded once per Pivot.

2.2.7.2 Linear Combinations

Each row T_i in T is in the *row span* of the constraints that defined the auxiliary variables ($T_i \cdot \mathcal{X} = 0$). A row T_i is in the row-span of A if it is the sum of rows in A scaled by a vector of constants, y,

$$T_i = \sum y_j \, A_j \, .$$

This immediately follows from the construction of T and its modification by pivoting which is implemented using scaled row addition (Fig. 2.10). This is often equivalently written using matrix multiplication as $y\,A$ where y is an n-dimensional row vector. The coefficients on the auxiliary variables will be able to tell us exactly the linear combination of the initial rows to form T_i. Any row $T_i \cdot \mathcal{X} = 0$ can be broken down into

$$\sum_{j \notin \mathrm{Aux}} T_{i,j} x_j + \sum_{s \in \mathrm{Aux}} T_{i,s} x_s = 0$$

Each auxiliary variable x_s is defined by a row A_s of the form $-x_s + \sum A_{s,j}\, x_j = 0$ where each x_j is structural. As each $-x_s$ only initially appears on its own row, $T_i \cdot \mathcal{X}$ must be the sum of these initial constraints.

Lemma 2.16. *For any row in T with index i, $T_i = y\,A$ where y is an n-dimensional row vector, $y_s = -T_{i,s}$ when x_s is auxiliary and y is 0 everywhere else.*

Proof. This property is clearly true for A and initially $T = A$. We now show that this property is preserved by induction on scaled row addition. Let $T'_j = T_j + \alpha T_i$ for some $\alpha \in \mathbb{R}$. Then by the inductive hypothesis there are z and z' such that $T_j = z\,A$ and $T_i = z'\,A$.

$$T'_j = T_j + \alpha T_i = z\,A + \alpha z'\,A = (z + \alpha z')\,A$$

Let $y = z + \alpha z'$. So $T'_j = y\,A$. For all entries where x_k is a structural variable, $y_k = 0$ as $z_k = z'_k = 0$. For all entries where x_s is a auxiliary variable, the new coefficient for x_s, $T'_{j,s}$, is equal to $T_{j,s} + \alpha T_{i,s}$. By the inductive hypothesis, $z_s = -T_{j,s}$ and $z'_s = -T_{i,s}$, so $y_s = -(T_{j,s} + \alpha T_{i,s})$. $\square$

Let $\mathfrak{R}_i$ be the set of constraints for the auxiliary variables that appear on T_i,

$$\mathfrak{R}_i = \{A_s \cdot \mathcal{X} = 0 | s \in \mathrm{Aux}, T_{i,s} \neq 0\} \, . \tag{2.21}$$

Corollary 2.17. $\mathfrak{R}_i \models_{\mathbb{R}} T_i \cdot \mathcal{X} = 0$.

Corollary 2.18. *If T_i is non-zero, then there exists some auxiliary variable x_s such that the coefficient of x_s is not zero and the length of the row is at least 2.*

Proof. Let y be the vector from Lemma 2.16, i.e. $T_i = y\,A$ and if $y_s \neq 0$, then $y_s = -T_{i,s}$ and x_s is auxiliary. Suppose T_i is non-zero. Then y is non-zero and $y_s = -T_{i,s} \neq 0$ for some auxiliary variable x_s. Suppose for contradiction that the length of T_i is less than 2. Therefore the only entry on T_i is $T_{i,s}$. By construction A_s, contains an entry for least one structural variable, $A_{s,j} \neq 0$. As $T_{i,j} = 0 \neq y_s\,A_{s,j}$, there must be some other $y_k \neq 0$. But then $T_{i,k}$ is non-zero, and the length of T_i is at least 2. □

2.2.7.3 Complexity

This section will show that the memory required to store the assignment α and the tableau T is polynomial in the size of the input. The complexity of a pivot is *strongly polynomial*. The core requirements for being strongly polynomial are that the number of arithmetic operations is bounded by the number of numeric constants in the input and that the intermediate numbers do not grow too fast. The number of arithmetic operations per pivot is naively $O(n^2)$. More challenging to see is that the number of bits required to represent each coefficient in T is polynomially bounded by the number of bits required to represent A.

Tableau form can be thought of as the result of Gaussian elimination for a particular variable order. An alternative view of a pivot is to take a matrix T that is reduced by Gaussian elimination under a variable order, and to change the variable order and reduce the matrix by Gaussian elimination under this new variable order. Schrijver gives a proof of the polynomial space bound of the size of rational coefficients by Gaussian elimination [98, Theorem 3.3]. (See this proof for a complete definition of the complexity bounds discussed in this section.) We show that each matrix T which is the result of a sequence of pivoting operation is identical to the result of some Gaussian elimination (up-to permutation). Hence all of the arithmetic operations remain strongly polynomial. The following lemma connects the tableau form of T to the Gaussian elimination of some permutation of A.

Lemma 2.19. *There exists a permutation matrix ρ such that applying Gaussian elimination to $\rho\,A\,\rho$ results in a matrix G such that $G = -\rho T \rho$.*

Proof. See Lemma A.5 in the appendix for a proof. □

Theorem 2.20. *The size of T and α is polynomial in the size of $\Phi_{\mathcal{A}}$ where $\Phi_{\mathcal{A}}$ is the set of all atoms in the input formula ϕ and lemmas sent to the SAT solver.*

Proof. See Lemma A.6 in the appendix for a proof. □

Remarks Many authors prefer a representation of T such that its dimensions are $m \times (m+N)$ where m is the number of auxiliary variables and N is the number of structural variables. This tighter representation places more emphasis on relative dimensions of T. However, due to the use of the sparse matrix data structures, the size of the matrix is dictated by the number of non-zero coefficients and so rows of 0s add negligible overhead. By having the tableau T be $n \times n$, a layer of indirection is avoided. (We also make use of square matrices in Chapter 3.)

2.2.8 Active Bounds and Row Based Inference

This section describes a novel method for efficiently detecting conflicts and propagations on the rows of the tableau. This is enabled by tracking variable statuses and aggregating these values across the rows and columns in T.

For a vector v in $\mathbb{R}^n$, let $\mathcal{L}_{act}(v)$ and $\mathcal{U}_{act}(v)$ be the sets of indices of variables whose lower bounds and upper bounds respectively are *active* in constraining the minimization of $\sum v_k a_k$: A bound is active if the assignment is equal to the bound.

$$\begin{aligned} \mathcal{U}_{act}(v) &= \{k | v_k < 0, \mathrm{Status}(x_k) \in \{\texttt{AT\_UB}, \texttt{FIXED}\}\} \\ \mathcal{L}_{act}(v) &= \{k | v_k > 0, \mathrm{Status}(x_k) \in \{\texttt{AT\_LB}, \texttt{FIXED}\}\} \end{aligned} \tag{2.22}$$

Note that for efficiency v_k can be treated as its sign function, $\mathrm{sgn}(v_k)$.

We use the statuses to track how close a row in T is to being either minimized or maximized. Minimizing the row T_i corresponds to finding a feasible assignment that minimizes the value of $\sum T_{i,k} a_k$. This is not yet particularly interesting as the minimum value must be 0 ($Ta = 0$); however, it will become interesting once one or more variables are excluded from the sum. To accomplish this, we track the cardinality of the four sets

$$\mathcal{U}_{act}(+T_i), \quad \mathcal{L}_{act}(+T_i), \quad \mathcal{U}_{act}(-T_i) \quad \text{and} \quad \mathcal{L}_{act}(-T_i)$$

for each basic variable, $i \in \mathcal{B}$. Let e_m be the unit vector for index m i.e. a vector that is 0 everywhere except at index m where it is 1. Let $\mathrm{mask}_{i,m}$ be an n dimensional vector that is row i excluding $T_{i,m}$:

$$\mathrm{mask}_{i,m} = T_i - T_{i,m} e_m.$$

Then given m, $\mathrm{sgn}(T_{i,j})$, $\sigma = \pm 1$, $|\mathcal{L}_{act}(\sigma T_i)|$, and $|\mathcal{U}_{act}(\sigma T_i)|$, we can compute in

$O(1)$-time the cardinalities $|\mathcal{L}_{act}(\sigma\,\mathrm{mask}_{i,m})|$ and $|\mathcal{U}_{act}(\sigma\,\mathrm{mask}_{i,m})|$ using

$$\begin{aligned}|\mathcal{L}_{act}(\sigma\,\mathrm{mask}_{i,m})| &= |\mathcal{L}_{act}(\sigma T_i - \sigma T_{i,m} e_m)| \\ &= |\mathcal{L}_{act}(\sigma T_i)| - \begin{cases} 1 & -\sigma\,\mathrm{sgn}(T_{i,m}) > 0, a_m = l_m \\ 0 & \text{otherwise.}\end{cases}\end{aligned}$$

The computation for $\mathcal{U}_{act}(\sigma\,\mathrm{mask}_{i,m})$ is similar. Intuitively, these cardinalities are the active variable counts for the upper and lower bounds excluding exactly the variable m from minimizing the sum, $\sigma T_i \cdot a$. The constant σ simply selects either maximization or minimization. The following mechanism then provides an efficient means for propagation and conflicts when all variables excluding x_m on T_i have the appropriate bound.

Let $\mathfrak{L}(v)$ be a set of constraints corresponding to the lower bounds for the indices in $\mathcal{L}_{act}(v)$, and $\mathfrak{U}(v)$ be the corresponding set for u and $\mathcal{U}_{act}(v)$. Note that

$$\begin{aligned}\mathfrak{L}(\sigma\,\mathrm{mask}_{i,m}) &= \{x_k \geqslant_\delta l_k | k \in \mathcal{L}_{act}(\sigma\,\mathrm{mask}_{i,m})\} \\ \mathfrak{U}(\sigma\,\mathrm{mask}_{i,m}) &= \{x_k \leqslant_\delta u_k | k \in \mathcal{U}_{act}(\sigma\,\mathrm{mask}_{i,m})\}\end{aligned}$$

Lemma 2.21. *Suppose* $T_{i,m} \neq 0$, $|\mathcal{L}_{act}(\sigma\,\mathrm{mask}_{i,m})| + |\mathcal{U}_{act}(\sigma\,\mathrm{mask}_{i,m})| + 1 = \|T_i\|_s$, *and (I3) and (I4) hold. Then the following must also hold:*

$$\mathfrak{L}(\sigma\,\mathrm{mask}_{i,m}) \cup \mathfrak{U}(\sigma\,\mathrm{mask}_{i,m}) \cup \mathfrak{R}_i \models_{\delta\mathbb{R}} -\sigma T_{i,m} x_m \geqslant_\delta -\sigma T_{i,m} a_m$$

Proof. This directly follows from applying Corollary 2.15 for z being σT_i and the sets $\mathcal{L}$ and $\mathcal{U}$ being $\mathcal{L}_{act}(\sigma\,\mathrm{mask}_{i,m})$ and $\mathcal{U}_{act}(\sigma\,\mathrm{mask}_{i,m})$ respectively. (See Lemma 2.16 for a description of computing the vector y.) □

Lemma 2.21 states that if all variables appearing on the row T_i (excluding x_m) are assigned to their upper or lower bounds, then x_m is currently minimized and $x_m \geqslant a_m$ [or maximized and $x_m \leqslant a_m$].

We incrementally track the cardinality of the 4 sets $\mathcal{L}_{act}(\pm T_i)$ and $\mathcal{U}_{act}(\pm T_i)$ for all $i \in \mathcal{B}$. Let $\mathtt{h}_{\sigma,i}$ denote the cardinality of $\mathcal{L}_{act}(\sigma T_i)$ for $\sigma = \pm 1$, and $\mathtt{g}_{\sigma,i}$ denote the cardinality of $\mathcal{U}_{act}(\sigma T_i)$ for all $i \in \mathcal{B}$. The summary of all 4 cardinalities

```
1: procedure SgnChange($i, j, \sigma, \sigma'$)
2:     $r_u \leftarrow$ if Status$(x_j) \in \{\texttt{AT\_UB}, \texttt{FIXED}\}$ then 1 else 0
3:     $r_l \leftarrow$ if Status$(x_j) \in \{\texttt{AT\_LB}, \texttt{FIXED}\}$ then 1 else 0
4:     if $\sigma \neq 0$ then
5:         $h_{\sigma,i} \leftarrow h_{\sigma,i} - r_l$
6:         $g_{\sigma,i} \leftarrow g_{\sigma,i} - r_u$
7:     if $\sigma' \neq 0$ then
8:         $h_{\sigma',i} \leftarrow h_{\sigma',i} + r_l$
9:         $g_{\sigma',i} \leftarrow g_{\sigma',i} + r_u$
```

Figure 2.11: Update the bound count when the sign of $T_{i,j}$ changes from σ to σ'

```
Require: $T_{i,j} \neq 0$
1: procedure StatusChange($T_{i,j}, st, st'$)
2:     $\sigma \leftarrow \mathrm{sgn}(T_{i,j})$
3:     $h_{\sigma,i} \leftarrow h_{\sigma,i} - ($if $st \in \{\texttt{AT\_LB}, \texttt{FIXED}\}$ then 1 else 0$)$
4:     $g_{\sigma,i} \leftarrow g_{\sigma,i} - ($if $st \in \{\texttt{AT\_UB}, \texttt{FIXED}\}$ then 1 else 0$)$
5:     $h_{\sigma,i} \leftarrow h_{\sigma,i} + ($if $st' \in \{\texttt{AT\_LB}, \texttt{FIXED}\}$ then 1 else 0$)$
6:     $g_{\sigma,i} \leftarrow g_{\sigma,i} + ($if $st' \in \{\texttt{AT\_UB}, \texttt{FIXED}\}$ then 1 else 0$)$
```

Figure 2.12: Update the bound count of a row when the status of a variable changes.

is below:

$$\begin{aligned}
h_{+1,i} &= \left|\left\{T_{i,j} > 0, a(x_j) = l(x_j)\right\}\right| \\
h_{-1,i} &= \left|\left\{T_{i,j} < 0, a(x_j) = l(x_j)\right\}\right| \\
g_{+1,i} &= \left|\left\{T_{i,j} < 0, a(x_j) = u(x_j)\right\}\right| \\
g_{-1,i} &= \left|\left\{T_{i,j} > 0, a(x_j) = u(x_j)\right\}\right|
\end{aligned}$$

Note that if $T_{i,j} \neq 0$, then x_j participates against either the count $h_{+1,i}$ or $h_{-1,i}$. To track these cardinalities incrementally, we add callback functions that are called whenever the sign of $T_{i,j}$ changes or Status(x_j) for $T_{i,j} \neq 0$ changes. The procedure SgnChange in Fig. 2.11 takes as input variable ids i and j, the previous sign of $T_{i,j}$, s and its current sign s' and adjusts $h_{\pm 1,i}$ and $g_{\pm 1,i}$. Fig. 2.13 gives a version of the update procedure in Fig. 2.1 that calls SgnChange. The procedure StatusChange in Fig. 2.12 takes as input the entry for the coefficient $T_{i,j}$, the previous status of x_j, $status$ and the current status $status'$ and adjusts $|\mathcal{L}_{act}(\pm T_i)|$ and $|\mathcal{U}_{act}(\pm T_i)|$ accordingly.

```
Require: $\delta \neq 0, x_j \in \mathcal{N}$
1: procedure UPDATE-STATUSCHANGE(j, $\delta$, STATUSCHANGE)
2:     $a_j \leftarrow a_j + \delta$
3:     for all $i$ such that $T_{i,j}$ do
4:         pre $\leftarrow$ Status$(x_i)$
5:         $a_i \leftarrow a_i + T_{i,j}\delta$
6:         post $\leftarrow$ Status$_i$
7:         STATUSCHANGE ($T_{i,j}$, pre, post)
```

Figure 2.13: An update procedure with STATUSCHANGE.

2.2.9 Conflicts

We will build upon the bookkeeping scheme presented in the previous section to identify conflicts on rows in constant time.

Chapter 3 discusses a variant of simplex that minimizes the sum of infeasibilities. We borrow from this chapter the notion of *the direction of violation* for a variable. When a variable's current assignment exceeds its upper bound, the value of the variable must be smaller in all feasible assignments (if one exists), i.e. $+1 \cdot x_i$ must be minimized to find a feasible assignment. Similarly $-1 \cdot x_i$ must be minimized when a variable's current assignment is below its lower bound. If a variable is between its bounds, its assignment may not have to change, and for conformity, $0 \cdot x_i$ can be trivially minimized. This coefficient is formalized as the direction of violation.

$$\lambda_i = \begin{cases} +1 & a_i > u_i \\ -1 & a_i < l_i \\ 0 & \text{otherwise} \end{cases}$$

We will revisit λ_i in Section 3.2. Whenever $\lambda_i \neq 0$, x_i must be basic by the invariant (I2), and it must also not be equal to its bounds so the active sets on the row T_i are the same as on $\text{mask}_{i,i}$. Due to the fact that if $\lambda_i \neq 0$, these basic variable x_i cannot be equal to its bound, we can conclude that:

$$\begin{aligned}
\mathcal{L}_{act}\,(\lambda_i\,\text{mask}_{i,i}) &= \mathcal{L}_{act}(\lambda_i T_i) \\
\mathcal{U}_{act}\,(\lambda_i\,\text{mask}_{i,i}) &= \mathcal{U}_{act}(\lambda_i T_i) \\
\mathfrak{L}\,(\lambda_i\,\text{mask}_{i,i}) &= \mathfrak{L}(\lambda_i T_i) \\
\mathfrak{U}\,(\lambda_i\,\text{mask}_{i,i}) &= \mathfrak{U}(\lambda_i T_i)
\end{aligned}$$

If $\lambda_i = 0$, the above sets are empty (as $\lambda_i T_i$ is the all 0 vector). Let $\text{VC}(k)$ be the

```
1: procedure CHECKBASICVARIABLESFORCONFLICTS($B \subseteq \mathcal{B}$)
2:     for all $i \in B$ do
3:         if $\lambda_i \neq 0$ then
4:             if $h_{\lambda_i,i} + g_{\lambda_i,i} + 1 = \|T_i\|_s$ then
5:                 $\langle S, c_i \rangle \leftarrow$ STRENGTHEN($\mathfrak{L}(\lambda_i T_i) \cup \mathfrak{U}(\lambda_i T_i) \cup \{VC(i)\}$)
6:                 $S \vdash \neg c_i$
7:                 $\{VC(i), \neg c_i\} \xrightarrow{\text{REGRESS2ASSERTIONS}^*} \mathcal{A}'$
8:                 add to output stream (Conflict $\mathcal{A}'$)
```

Figure 2.14: A simple procedure for checking whether some basic variable in a list is in conflict.

violated constraint forcing the minimization of $\lambda_k x_k$.

$$VC(k) = \begin{cases} c : x_k \leqslant u_k & a_k > u_k \\ c : x_k \geqslant l_k & a_k < l_k \\ \text{undefined} & \text{otherwise} \end{cases}$$

Using these notions of violated constraints, invariants (I3) and (I4), and Lemma 2.21, we can devise a constant time check if row i contains a conflict.

Lemma 2.22. *Suppose* $\lambda_i = \pm 1$, $h_{\lambda_i,i} + g_{\lambda_i,i} + 1 = \|T_i\|_s$, *(I3) and (I4) hold. Then*

$$\mathfrak{L}(\lambda_i T_i) \cup \mathfrak{U}(\lambda_i T_i) \cup \mathfrak{R}_i \models_{\mathbb{R}} \neg\, VC(i)$$

Proof. Assume that $\lambda_i \neq 0$ and $h_{\lambda_i,i} + g_{\lambda_i,i} + 1 = \|T_i\|_s$ hold as well as the invariants (I3) and (I4). The variable x_i must be basic and the coefficient of x_i on its row must be $T_{i,i} = -1$. The union of $\mathcal{L}_{act}(\lambda_i T_i)$ and $\mathcal{U}_{act}(\lambda_i T_i)$ must not be empty. Then by Lemma 2.21 (with $\sigma = \lambda_i$), the constraints in $\mathfrak{L}(\lambda_i \operatorname{mask}_{i,i})$, $\mathfrak{U}(\lambda_i \operatorname{mask}_{i,i})$ and $\mathfrak{R}_i$ entail that

$$-\lambda_i T_{i,i} x_i \geqslant_\delta -\lambda_i T_{i,i} a_i \implies \lambda_i x_i \geqslant_\delta \lambda_i a_i \ \ (\text{as } T_{i,i} = -1).$$

Suppose that $\lambda_i = -1$. Then $x_i \leqslant_\delta a_i$ is entailed, $a_i <_\delta l_i$, and the negation of $VC(i)$ is equivalent to $x_i <_\delta l_i$. Hence, $\neg\, VC(i)$ is entailed as well. The proof for $\lambda_i = +1$ is analogous. □

Remark. The inference rules used by LowerConflict(x_i) and UpperConflict(x_i) in the abstract decision procedure in section 2.1.4 are subsumed by 2.22.

Any list of basic variable indices B can then be checked for conflicts in time $O(|B|)$. See Figure 2.14 for a sketch of an algorithm to accomplish this.

Lemma 2.23. *The conflict* $\mathfrak{L}(\lambda_i T_i) \cup \mathfrak{U}(\lambda_i T_i) \cup \mathfrak{R}_i \cup \{VC(i)\}$ *is minimal.*

Proof. We now show that for all constraints in the conflict there exists an assignment that satisfies all but that constraint. Let $\Delta = u_i - a_i$ if $a_i > u_i$ or $\Delta = l_i - a_i$ otherwise. For any j such that $T_{i,j} \neq 0$, we will define an assignment $a^{(j)}$. Let $a^{(i)} = a$. For any non-basic variable j on the T_i, let $a^{(j)}$ be the assignment resulting from performing the pivot and update operation $\textsc{PivotAndUpdate}(i, \Delta, j)$. Each $a^{(j)}$ satisfies $Ta^{(j)} = 0$ and the bounds for every variable in the conflict except the one for x_j. Thus all bounds on variables must be in the conflict.

Each equality in $\mathfrak{R}_i$ corresponds to the defining equality of an auxiliary variable x_s. By setting the value of $a^{(s)}$ to the bound on x_s in the conflict, we get a new assignment. This new assignment satisfies all of the equalities in $\mathfrak{R}_i$ except the one corresponding to x_s and all of the bounds. □

The conflicts coming from Lemma 2.22 always use the strongest available constraints on the variables in the set of assertions,

$$\mathfrak{L}(\lambda_i T_i) \cup \mathfrak{U}(\lambda_i T_i) \cup \mathfrak{R}_i \cup \{VC(i)\}$$

This is not desirable. Consider the sequence of assertions:

$$x_1 < x_2, x_1 \leqslant x_2, x_2 < x_3, x_2 \leqslant x_3, \ldots, x_{N-1} < x_N, x_{N-1} \leqslant x_N, x_N \leqslant x_1, x_N < x_1$$

In problems with many variable equality atoms $x = y$, atoms such as $x \leqslant y$ and $x \geqslant y$ arise. (See Section 2.2.10.)[13] The conflict generated by Lemma 2.22 corresponds to the conflict

$$C_1 = \{x_1 < x_2, x_2 < x_3, \ldots, x_{N-1} < x_N, x_N < x_1\}$$

instead of the stronger set of constraints C_2 with only a single strict inequality

$$C_2 = \{x_1 < x_2, x_2 \leqslant x_3, \ldots, x_{N-1} \leqslant x_N, x_N \leqslant x_1\}.$$

Both of the lemmas $\neg C_1$ and $\neg C_2$ would force the theory solver to backtrack but $\neg C_2$ is more restrictive for future search. Let $U = \{x_i < x_{i+1} \implies x_i \leqslant x_{i+1}\}$ or the set of unate implications.

$$U \cup \{\neg C_2\} \models \neg C_1 \qquad \text{but} \qquad U \cup \{\neg C_1\} \text{ does not entail } \neg C_2. \tag{2.23}$$

Note the use of first-order entailment $\models$ in 2.23. Essentially, there is additional surplus in the transition from Lemma 2.21 to Lemma 2.22 that we have not yet used advantageously. This subsection defines a novel method for heuristically using this surplus.

[13] Due to unate propagation (Section2.2.14), both the strict and non-strict inequalities $x \leqslant c$ and $x < c$ are often asserted.

Let the initial surplus of the conflict `surplus` be either $a_i - u_i$ or $l_i - a_i$. There are many candidate choices for how to strengthen the conflict. CVC4 uses a greedy algorithm to weaken the bounds on the variables participating in the conflict to find conflicts with a smaller `surplus`. The algorithm iterates over the constraints in $\mathfrak{L}(\lambda_i T_i) \cup \mathfrak{U}(\lambda_i T_i)$. For each constraint $c : x_j \geqslant d$ in the set $\mathfrak{L}(\lambda_i T_i)$, let c' be the next strictly weaker [asserted] bound on x_j, i.e. $c' : x_j \geqslant d'$ and $d > d'$. If $\texttt{surplus} > \left|\frac{d-d'}{T_{i,j}}\right|$, c is relaxed to c' in the conflict and surplus is correspondingly updated ($\texttt{surplus} \leftarrow \texttt{surplus} - \left|\frac{d-d'}{T_{i,j}}\right|$). This continues until c cannot be relaxed any further. The algorithm then attempts to greedily relax the constraints on the next variable. We call the resulting set of constraints the strengthening of $\mathfrak{L}(\lambda_i T_i) \cup \mathfrak{U}(\lambda_i T_i) \cup \{\mathrm{VC}(i)\}$. We denote the strengthening of a conflict as $c_i \cup S$ where c_i is the resulting constraint on $\mathrm{VC}(i)$ and S contains the remaining constraints. The negation $\neg c_i$ can be deduced to follow from the rest of the constraints in the explanation S and $\mathfrak{R}_i$. (The constraint for c_i may also be strengthened while there is additional surplus.)

This set of constraints is not yet a valid conflict as it contains purely internal x_s auxiliary variables. These can conceptually be removed by replacing each x_s variable using the equalities in $\mathfrak{R}_i$ ($-x_s + \sum_{i=1}^{N} x_i = 0$). After substitution by the equalities in $\mathfrak{R}_i$ and rewriting, the result would be a valid theory lemma. Note though that all of these constraints in S have been asserted to the theory. These are equivalent to the assertions at the assertion indexes for the constraint. By (I1), the explanation for $\neg c_i$ is not set, and $S \vdash \neg c_i$ may be set. Then repeatedly applying REGRESS2ASSERTION (2.15) to $\{\neg c_i, \mathrm{VC}(i)\}$ results in a subset of the assertions ($\mathcal{A}'$),

$$\{c_i, \mathrm{VC}(i)\} \xrightarrow{\textsc{Regress2Assertions}^*} \mathcal{A}'$$

This set of assertions is a theory valid conflict, $\models_{\mathbb{R}} \neg\mathcal{A}'$.

2.2.10 Adding Disequalities Relations

The previous section removed all atoms of the form $t = d$ by rewriting these to $t \leqslant d \wedge t \geqslant d$ for a linear term t. Clearly, asserting equalities via a procedure ASSERTEQUALITY can be handled for a variable x as a synonym for running ASSERTUPPER($x \leqslant d$) and ASSERTLOWER($x \geqslant d$) internally.[14] The reason these are removed in the abstract presentation (Section 2.1) is to avoid handling negations of such atoms, $x \neq d$. Disequalities are unique in that these are the first non-convex constraints we have encountered. While there exist algorithms for handling disequalities natively [97], these tend to have unclear advantages over simpler techniques.

[14]The coefficient for δ is 0 in the $\delta\mathbb{Q}$.

```
1: procedure SplitDisequalities
2:     for all $c : x_j \neq d + 0\delta \in \mathcal{D}$ do
3:         if $a_j \neq d$ and c is not split then
4:             $t \leftarrow x_j + A_j$ // Cancel $x_j$ from the row
5:             output lemma $(t = d \iff (t \leqslant d \wedge t \geqslant d))$ to the SAT solver
6:             // this is equivalent to $t \neq d \iff (t > d \vee t < d)$
7:             mark c as split
```

Figure 2.15: Lazy splitting for disequalities.

CVC4 handles disequalities by lazily introducing the previous rewrite using splitting-on-demand lemmas (Fig. 2.15). As disequality constraints come in, the AssertDisequality$(c : t \neq d)$ procedure merely adds c to a context dependent list of disequalities, $\mathcal{D}$. On full effort checks, all of the disequality constraints that are not satisfied by the current assignment, a, are split by sending lemmas to the SAT solver. If no splits are issued and the SAT solver is at full effort, each constraint $t \neq d$ in $\mathcal{D}$ is satisfied by a as either previously split lemmas satisfy either $t < d$ or $t > d$, or the $a_{\mathrm{NodeToId}(t)} \neq d$.[15]

2.2.11 Computing Models with Disequalities

To compute models with disequalities, we use the algorithm for selecting an order-preserving range $(0, \alpha')$ given in Section 2.2.1.4. Let Q be the set of $\delta\mathbb{R}$ constants for all of the bounds, the disequalities, and the variable assignments.

$$Q = \{a_i\} \cup \{l_i | l_i > -\infty\} \cup \{u_i | u_i < +\infty\} \cup \{d | c : x \neq d \in \mathcal{D}\}$$

Compute α' as described in Section 2.2.1.4. Then by Lemma 2.9 and $\beta \in (0, \alpha')$ preserves the order of the elements in Q. The interpretation $M_{a_\delta,\beta}$ then satisfies the input assertions.[16]

Lemma 2.24. *If* $Ta =_\delta 0$, $l \leqslant_\delta a \leqslant_\delta u$, *and* $a \models_{\delta\mathbb{R}} \mathcal{D}$, *then*

$$M_{a_\delta,\beta} \models_{\mathbb{R}} A\,\mathcal{X} = 0 \wedge l \leqslant \mathcal{X} \leqslant u \wedge \mathcal{D}.$$

Proof. As β preserves the orders between all of the elements of Q, the satisfaction of each of the individual constraints is satisfied translates from the $\delta\mathbb{R}$ level to the reals. □

[15] This is subtly incompatible with CVC4's theory sharing and lemma generation infrastructure. See Section A.1.2 in the appendix.

[16] The original technical report [47] gives a constructive algorithm $O(n)$ time algorithm for selecting β to satisfy constraints of the form $l \leqslant \mathcal{X} \leqslant u$.

Remarks. In $M_{a_\delta,\beta}$, no variables that are disequal according to a become equal. This also makes this selection method appropriate for model construction with theory combination.

2.2.12 Termination with Heuristic Variable Selection

Each round of Simplex selects a basic variable x_i to leave $\mathcal{B}$ by pivoting it with a non-basic variable x_j that enters $\mathcal{B}$. To ensure termination of SIMPLEX-FORSMT, it suffices to select for the leaving variable the minimum $i \in E$ and for the entering variable the minimum $j \in \texttt{entering}$ (lines 4, 7 and 10 in Figure 2.7). This is one of the many variants of Bland's rule for constructing a terminating simplex implementation.

Lemma 2.25. *Any execution of* SIMPLEXFORSMT *reaches only a finite number of distinct* a*.*

Proof. Each variable is either basic or non-basic. There are $\binom{n}{|B|}$ candidate sets of selecting the basic variables. Each non-basic variable x_j is assigned to either a bound (l_j or u_j), or to the assignment it initially had upon starting this Simplex search. The assignment to the basic variables is determined by the non-basic variables. □

Corollary 2.26. SIMPLEXFORSMT *does not terminate iff there exists a cycle in the pairs of the set of basic variables and the assignment* $\langle \mathcal{B}, a \rangle$*.*

As is shown in the proof of termination of SIMPLEXFORSMT [47, Theorem 1], Bland's rule ensures that the pairs $\langle \mathcal{B}, a \rangle$ do not cycle. Any variable selection rule that eventually converges to Bland's rule must also terminate. All of these heuristics can be described as selecting according to two total orders $\prec_{\texttt{entering}}$ and $\prec_{\texttt{leaving}}$ that are adapted after each selection and eventually converge to $\prec$. Figure 2.16 gives an refined version of DUALSELECT. This version additionally assumes $\forall x_i \in E : \texttt{leaving}_i \neq \emptyset$.

Lemma 2.27. *The procedure* SIMPLEXFORSMT *terminates if* $\prec_{\texttt{entering}}$ *and* $\prec_{\texttt{leaving}}$ *eventually converge to* $\prec$*.*

We now define the set of entering variables using the definition of the active constraints. The entering variables will be those without an active constraint on a row i. Recall the definition of $\mathcal{F}$ from Section 2.2.2, $\mathcal{F} = \{k | z_k \neq 0\} \setminus (\mathcal{L} \cup \mathcal{U})$. We generalize this in terms of the active variables $\mathcal{L}_{act}(v) \cup \mathcal{U}_{act}(v)$.

$$\mathcal{F}_{act}(v) = \{k | v_k \neq 0\} \setminus (\mathcal{L}_{act}(v) \cup \mathcal{U}_{act}(v))$$

```
Require: $E \neq \emptyset, \forall i \in E . h_{\lambda_i,i} + g_{\lambda_i,i} + 1 < \|T_i\|_s$
1: procedure DualSelect-HeuristicOrders
2:     select $i$ from $E$ to minimize $\prec_{\texttt{leaving}}$
3:     select $j$ from $\mathcal{F}_{act}(\lambda_i \, \text{mask}_{i,i})$ to minimize $\prec_{\texttt{entering}}$
4:     $\delta \leftarrow$ if $(a_i > u_i)$ then $(u_i - a_i)$ else $(l_i - a_i)$
5:     return $\langle i, \delta, j \rangle$
```

Figure 2.16: A pivot and update selection routine with heuristic variable orders.

We will call the set $\mathcal{F}_{act}(\lambda_i \, \text{mask}_{i,i})$ the *inactive* non-basic variables for the row i. These will be the candidate entering variables for the row i. It then follows from Lemma 2.22, that

$$\mathcal{F}_{act}(\lambda_i \, \text{mask}_{i,i}) \neq \emptyset \iff h_{\lambda_i,i} + g_{\lambda_i,i} + 1 < \|T_i\|_s .$$

The heuristic commonly used in implementations of SimplexForSMT for selecting the entering variable is to select the x_j with minimum column length $|(T^T)_j|$ with ties broken by $\prec$. This heuristic works quite well in practice but is not guaranteed to terminate. A simple means of ensuring termination is to count the number of heuristic selections and switch to Bland's rule once this passes a finite cap. This is roughly what is described in MathSat [60] and OpenSMT [22]. Another variant is to track how many times the variable x_i has left the basis in an execution of Simplex. If this count goes over a threshold for any variable, the procedure switches to Bland's rule. This strategy is used by default in Yices, Yices 2, and Z3 [39]. CVC4's heuristic is to to track how many times the variable x_i has left the basis in an execution of Simplex and if this count is over a threshold select the entering variable to be the least variable in $\mathcal{F}_{act}(\lambda_i \, \text{mask}_{i,i})$. The intention of this rule is to try to eliminate the small cycles that appear using this heuristic, but without fully switching to Bland's rule.

The most common way of selecting the leaving variable is to select the minimum i in E to leave. Griggio describes in his thesis a heuristic for selecting the leaving variable in E that violates its bound the most [60]. After a finite number of rounds, the heuristic switched to Bland's rule. CVC4 and MathSAT5 implement this heuristic.

2.2.13 Theory Solver

The techniques given throughout this section can now be combined to form a decision procedure for `QF_LRA`. Figure 2.18 gives a version of PivotAndUpdate that appropriately reports the changes of the signs of coefficients and statuses. Figure 2.17 builds a new version of the main simplex loop using the new

```
1: procedure SimplexForSMTCheck
2:     checkBasicVariablesForConflicts (E)
3:     loop
4:         if (Conflict C) $\in$ outputStream then
5:             return Conflict
6:         else if $E = \varnothing$ then
7:             return (Sat $a$)
8:         else
9:             $\langle i, \delta, j \rangle \leftarrow$ DualSelect-HeuristicOrders()
10:            PivotAndUpdate-SC ($i, j, \delta$, StatusChange, SgnChange)
11:            checkBasicVariablesForConflicts ($\{k | T_{k,i} \neq 0\} \cap E$)
```

Figure 2.17: A refined version of the main loop of the SimplexForSMT check procedure.

```
1: procedure PivotAndUpdate-SC($i, j, \delta$, StatusChange, SgnChange)
2:     Pivot-SC ($i, j$, SgnChange)
3:     Update-StatusChange ($i, \delta$, StatusChange)
```

Figure 2.18: Pivot and update with sign and status changes.

pivot and update procedure and the procedure in Fig. 2.16 for selecting the entering and leaving variables. It additionally detects with amortized constant time overhead row conflicts over all rows.

Lemma 2.28. *Let Φ be the set of atoms in the input formula ϕ and Ψ be the set of arithmetic atoms in lemmas generated by other modules in CVC4. Then* SimplexForSMT *generates $O(|\Phi| + |\Psi|)$ new atoms by splitting equalities, and the SMT solver terminates if Ψ is finite and the other modules terminate.*

Theorem 2.29. *The SMT solver is a sound and complete decision procedure for* `QF_LRA`

```
1: procedure T-Check(effort)
2:     GetAssertionsOffStack ()
3:     if rrstatus = Unknown $\wedge$ (Conflict C) $\notin$ outputStream then
4:         SimplexForSMT
5:         if (Conflict C) $\notin$ outputStream then
6:             rrstatus $\leftarrow$ Sat
7:     if rrstatus = Sat and effort is full effort then
8:         SplitDisequalities ()
```

Figure 2.19: Complete SimplexForSMT theory check procedure.

using this theory solver.

Proof. If T-CHECK (full effort) returns no conflicts or lemmas, then the assignment a satisfies $Ta =_\delta 0$, $l \leqslant_\delta a \leqslant_\delta u$, and $a \models_{\delta\mathbb{R}} \mathcal{D}$. Then by Lemma 2.24, there exists a $\mathcal{T}_\mathbb{R}$ satisfying interpretation $M_{a,\beta'}$. Thus if the SMT solver terminates with **Sat** the input formula is satisfiable.

If the the SMT solver terminates with **Unsat**, then as all of the theory lemmas are emitted to the output stream during T-CHECK are $\mathcal{T}_\mathbb{R}$-valid, the input formula is unsatisfiable. The SMT solver terminates using this theory solver for `QF_LRA` as the theory solver introduces only a finite number of splits. □

2.2.14 Propagation

A theory solver may propagate to the SAT solver that the current set of assertions $\mathcal{A}$ entails a literal p known to both the SAT solver and the theory solver.

$$\mathcal{A} \models_\mathbb{R} p$$

The SAT solver may then assign the literal to a fixed value in the current sat-context-level. As this has the potential to cut down an exponential amount of search by the SAT solver, designing propagation schemes that balance doing an incremental amount of work with successfully deducing entailed literals are essential for designing theory solvers. The SAT solver lets the theory solver know the set of literals that are candidates for propagation via a preregistration call (see Sec. 1.5.6). The theory solver must be able to later explain any of the propagations it has made in scope of the current sat-context-level.

A *complete propagation rule* detects all possible propagations. Linear programming (Sec. 3.1) may be used as a complete propagation rule by maximizing and minimizing all variables at every round; however, this is expensive. Propagation in CVC4's $\mathcal{T}_\mathbb{R}$ theory solver focuses on two simple but incomplete rules: unate propagation and row propagation.

Unate propagation is the simple rule:

$$\frac{d, d' \in \mathbb{R} \qquad d \leqslant d'}{x \leqslant d \models_\mathbb{R} x \leqslant d'} \quad \frac{d, d' \in \mathbb{R} \qquad d \geqslant d'}{x \geqslant d \models_\mathbb{R} x \geqslant d'}$$

This form of propagation is done in $O(1)$ time via the datastructures in the global constraint manager. (See Sec. 2.2.4 for more on constraints.) The constraints for all variables are stored in a map sorted by their right-hand-side (the d values above). Once a constraint c has been asserted, a pointer to the next weakest constraint is either a forward or a back pointer in the tree. If this next element c' has no explanation, then c may be set as its explanation and c' may be propagated if it has been preregistered. This process continues until it finds some weaker c'' with an explanation. CVC4 additionally has a mode for eagerly instantiating all unate implications as lemmas before solving begins. This results in adding a linear number of binary clauses.

Row propagation uses the same tools as conflict generation. Given a row T_i, row propagation attempts to learn either an upper or a lower bound on a single variable x_j. We again use $\sigma = \pm 1$ to control whether an upper or lower bound is being learned. Row propagation uses σT_i to select $\mathcal{L}$ and $\mathcal{U}$ to exclude exactly j where $T_{i,j} \neq 0$ ($\mathcal{F} = \{j\}$). Lemma 2.12 can be applied to learn the inequality $-\sigma T_{i,j} x_j \geqslant_\delta \gamma$ in extended arithmetic. When γ is finite, this can entail any weaker bound in the system via unate propagation. (If γ is infinite, the inequality is trivially **true**.) When $-\sigma T_{i,j}$ is positive, the learned bound is a lower bound, and when $-\sigma T_{i,j}$ is negative, the learned bound is an upper bound. As there are $O(n^2)$ possible choices of i, j, and σ and each attempt takes roughly $O(n)$ time, the challenge is to judiciously attempt only interesting selections and filter out selections that cannot succeed.

The rest of this subsection examines a heuristic method for filtering these cases. (Readers are encouraged to review Lemma 2.12 and Section 2.2.8 before continuing.) During solving we additionally track the set of variables x_i such that the procedure $\textsc{AssertionCases}(x_i \bowtie d)$ has asserted a bound on it since the last round of row propagation. Call this set S. (S may be over-approximate so it does not have to be backtracked.) Using the same tools used to track the number of active variables on each row ($\mathtt{h}_{\sigma,i}$ and $\mathtt{g}_{\sigma,i}$), we additionally track how many variables on each row have any bound at all. We call these four quantities $\mathtt{H}_{\sigma,i}$ and $\mathtt{G}_{\sigma,i}$. For consistency, row propagation is only run whenever all $\mathcal{A}_0$ through $\mathcal{A}_{\mathtt{processedPos}}$ are known to be satisfied by the assignment a, i.e. `rrstatus` = **Sat** (see Sec. 2.2.5). Row propagation begins iterating over the columns of all $j \in S$ to collect a set of row indices R. Then for all $i \in R$ and $\sigma = \pm 1$, row propagation first checks if $\mathtt{H}_{\sigma,i} + \mathtt{G}_{\sigma,i} + 1 \geqslant \|T_i\|_s$. If not, propagation is not possible using this row. If $\mathtt{H}_{\sigma,i} + \mathtt{G}_{\sigma,i} + 1 = \|T_i\|_s$, there is a unique variable j on the row that does not have a bound (modulo σ). This is the only variable propagation possible on this row and σ. Otherwise, every variable has a bound (modulo σ) and is a candidate for propagation. We consider these two cases separately.

Suppose there is a unique j such that it does not have a relevant bound for the row σT_i. Let $\sigma_j = -\operatorname{sgn}(T_{i,j})\sigma$ for all j on T_i. The set $\mathcal{F} = \{j\}$ is the only variable left from $\mathcal{L}$ and $\mathcal{U}$. This selection defines the value of γ. The following steps are used to try to propagate on such a j.

- Any entailed propagation is satisfied by the current satisfying assignment a. So if a_i is equal to its current relevant bound, then no new bound may be learned. $O(1)$.

- Check if there exists a bound in the constraint database that is strictly weaker than a_j for x_j. Supposing $\sigma_j = 1$, this is a constraint $c : x_j \geqslant d$ and $a_i > d$. $O(\log n)$.
- Compute γ. All other variable have bounds so γ must be finite. By Lemma 2.12, $-\sigma T_{i,j} x_j \geqslant_\delta \gamma$ with the explanation for c is $\mathcal{A}'_\delta$. $O(n)$.
- Check if there exists a constraint in the constraint manager such that it is entailed by $-\sigma T_{i,j} x_j \geqslant_\delta \gamma$. Let c be the strongest such entailed constraint. Supposing $\sigma_j = 1$, c is a constraint $c : x_j \geqslant d$ and $\frac{\gamma}{-\sigma T_{i,j}} > d$. $O(\log n)$.
- If c is not already the lower or upper bound for x_j, then propagate c and set its explanation. $O(n)$.[17]

When $\mathtt{H}_{\sigma,i} + \mathtt{G}_{\sigma,i} = \|T_i\|_s$, all variables are candidate for propagation on row σT_i. We first compute $\Delta = \gamma$ for when $\mathcal{F} = \varnothing$, i.e. $\mathcal{L} \cup \mathcal{U} = \{j | T_{i,j} \neq 0\}$. As propagation is only applied when the system is consistent, $\Delta <_\delta 0$. The quantity of Δ allows for the efficient computation of the value for γ for every candidate $\mathcal{F} = \{j\}$, call this γ_j.

$$\gamma_j = \Delta - \begin{cases} \sigma T_{i,j} l_j & \sigma_j = 1 \\ \sigma T_{i,j} u_j & \sigma_j = -1 \end{cases}$$

We can then attempt each candidate j as was done for the unique case, but using the γ_j values. We therefore pay the $O(n)$ cost to compute the γ values only once per row instead of the naive $O(n^2)$ for the row for computing the γ values.

[17] Setting the explanation is $O(n)$; however, as this only applies to successful propagations, this is not considered burdensome.

Chapter 3

Simplex with Sum of Infeasibilities for SMT

This chapter gives a new theory solver for quantifier-free linear real arithmetic. This theory solver is built around the SIMPLEXFORSMT algorithm given by Dutertre and de Moura [47]. (See Chapter 2.) That algorithm works by performing a sequence of local optimization operations that select which pivoting operations to perform and relies on specific pivoting heuristics to search for a satisfying model or a conflict. Many pivot choices are possible and these choices can dramatically change the search for a solution. The heuristic pivot selection scheme that many SMT solvers use (Section 2.2.12) is based on local criteria and is potentially subject to cycling: it may return to the same basis state infinitely often. Solvers employ tactics to detect cycling, and slowly edge towards pivot-selection rules that guarantee termination, such as Bland's Rule [56, 98]. Unfortunately, Bland's rule may converge very slowly and is not effective on hard problems that require many pivots.

While the algorithm is generally efficient in practice on verification problems, its local pivoting heuristics can lead to slow convergence towards either a satisfying assignment or a conflict. In contrast to more traditional Simplex algorithms, SIMPLEXFORSMT does not perform global optimization. Dantzig originally developed the Simplex method in the late 1940s to solve logistical problems for the US Air Force by formulating the logistical problems as optimization problems [55]. The standard Simplex algorithm finds a solution that is "best" according to some criteria. This is made mathematically explicit by adding a linear objective function f that is to be minimized (or equivalently maximized). The linear constraints combined with a linear objective are called *Linear Programs* (LPs), and systems that solve them are called *LP solvers*. Throughout execution of the Simplex algorithm, the value of f never increases. As long as f strictly decreases, no cycling is possible. Thus, specialized techniques to prevent cycling are only required to break out of sequences of degenerate pivots, that is, pivots that do not change f. Procedures can then be designed around two different modes: a heuristic mode that is efficient in practice, and a mode for escaping cycling. The search may then be strongly biased towards looking for choices that decrease the value of the optimization function. This has the consequence of making every round more expensive by doing additional analysis, but potentially reducing the number of Simplex rounds before converging to a solution. Before SIMPLEXFORSMT, earlier simplex-based approaches for SMT used repeated optimization (via an algorithm like PRIMAL in Section 3.1) as constraints arrived [5, 43, 97].

This chapter proposes an adaptation of the sum-of-infeasibilities method from the Simplex literature in the context of SMT [19, 56]. We call this method SOISIMPLEX. Minimizing the sum-of-infeasibilities provides a witness function similar to f which accomplishes several things at once: it helps guide the search towards both models and conflicts; it prevents cycling; and it can be used to determine when to safely re-enable aggressive heuristics without losing termination. In other aspects, SOISIMPLEX is similar to the SIMPLEXFORSMT algorithm, providing similar features and having similar performance on many problems. However, its performance is noticeably better on certain problem instances that require many pivots.

The material of this chapter has previously been published in [73]. This chapter assumes familiarity with the abstract description of SIMPLEXFORSMT in Section 2.1 and selectively refers to some of the more advanced refinements discussed in Section 2.2. The rest of this chapter is organized as follows. Section 3.1 describes a naive traditional primal simplex optimization routine. Section 3.2 then describes the sum-of-infeasibilities algorithm. Empirical results and a detailed comparison to SIMPLEXFORSMT are given in Section 3.4.

3.1 Naive Primal Simplex

The classic problem in linear optimization is to find an assignment a that satisfies the linear equalities $Ta = 0$ and the bounds $l \leqslant a \leqslant u$, and that minimizes a linear function $f = \sum_{x_k \in \mathcal{X}} c_k x_k$. The problem can be solved with the PRIMAL Simplex algorithm shown in Figure 3.1. It is typical to assume that the algorithm is given an initial feasible assignment as input a' such that both $Ta' = 0$ and $l \leqslant a' \leqslant u$ are initially satisfied. This problem is often known as the Linear Programming problem.

The optimization function f is treated as a special additional variable

$$x_f = \sum_{x_k \in \mathcal{X}} c_k x_k. \tag{3.1}$$

```
procedure PRIMAL(f)
    while $\mathcal{F}_{act}(\tau_f) \neq \varnothing$ do
        $\langle i, j, \delta \rangle \leftarrow$ PRIMALSELECT()
        UPDATE$(j, \delta)$
        if $i \neq j$ then
            PIVOT$(i, j)$
    return $a(f)$
```

Figure 3.1: PRIMAL(f) with a generic selection routine.

```
procedure PRIMALSELECT
    $S \leftarrow \varnothing$
    for all $j \in \mathcal{F}_{act}(\tau_f)$ do
        $S \leftarrow S \cup \langle j, k \rangle$, where $\langle |\delta_B(j, k)|, k \rangle$ is minimal
    select $\langle j, k \rangle \in S$ minimizing $\left\langle -|\operatorname{sgn}(\delta_B(j, k))T_{f,j}|, j \right\rangle$ by $<_{lex}$
    return $\langle j, \delta_B(j, k), k \rangle$
```

Figure 3.2: PRIMALSELECT with a terminating variant of Dantzig's rule.

We treat x_f as a synonym for the variable with index 0 (x_0). We add a row and column for f to the matrix A which defines the auxiliary variables (Sec. 2.1.2). For clarity, we denote the coefficients on f's row as $A_{f,j} = c_j$, for $1 \leqslant j \leqslant n$, $A_{i,f} = 0$ for $1 \leqslant i \leqslant n$, and $T_{f,f} = -1$. To add the equality (3.1) as a row to T while maintaining tableau form, each basic variable x_k with c_k is canceled out of the T_f by adding $c_k T_k$ to (3.1) until all of the coefficients of basic variables are zero (as was done in Sec. 2.2.3). We can then treat x_f as an auxiliary, basic variable with no bounds. (Note that to instead maximize f with the same machinery, we simply minimize its negation $-f$.)

Every round of PRIMAL begins by checking whether or not f is currently at its minimum. This is done by looking at the assignments to each nonbasic variable on f's row. The value of x_j that minimizes f–call this ν_j–is u_j if $T_{f,j}$ is negative and l_j if $T_{f,j}$ is positive (ignoring other constraints). If $a_j = \nu_j$ for each nonbasic variable x_j on f's row (where $T_{f,j} \neq 0$), then the current value of f, a_f, must be the minimum because we can prove $x_f \geqslant a_f$ as follows:[1]

$$\begin{aligned} x_f &= \sum_{\tau_{f,j}>0} T_{f,j} x_j + \sum_{\tau_{f,k}<0} T_{f,k} x_k \\ &\geqslant \sum_{\tau_{f,j}>0} T_{f,j} l_j + \sum_{\tau_{f,k}} T_{f,k} u_k = \sum_{\tau_{f,j}>0} T_{f,j} a_j + \sum_{\tau_{f,k}<0} T_{f,k} a_k = a_f. \end{aligned} \tag{3.2}$$

[1] This is an application of Corollary 2.15.

The search can then terminate. Otherwise, there is some x_j on f's row s.t. $a_j \neq \nu_j$, and it is unclear whether a_f is at a minimum. By trying to change a_j for these x_j, we can at the same time hunt for an assignment that decreases a_f and search for a proof of optimality.

It is convenient in the subsequent discussion to use the matrix τ obtained from the T such that non-basic variables have coefficient 1 on their rows and the rows for the basic variables only have coefficients for the non-basic variables. This leads to τ being defined as $T + I$ (where I is the identity matrix). Note that on the diagonal, $\tau_{i,i} = 0$ for $i \in \mathcal{B}$ and $\tau_{j,j} = 1$ for $j \in \mathcal{N}$ (off the diagonal, $\tau_{i,j} = T_{i,j}$). We refer to the i'th row of τ as τ_i and to the entry in row i and column j of τ as $\tau_{i,j}$.[2]

The τ matrix simplifies several notions including defining the variables that restrict the minimization of f. Borrowing the notion of active variables from Section 2.2.12, the non-basic variables against their bounds are active. The indices of the inactive variables on f's row whose assignments do not restrict the minimization of f are $\mathcal{F}_{act}(\tau_f)$. Recall from Sections 2.2.8 and 2.2.12 the definition of $\mathcal{L}_{act}$, $\mathcal{U}_{act}$ and $\mathcal{F}_{act}$. The indices in $\mathcal{L}_{act}(\nu)$ and $\mathcal{L}_{act}(\nu)$ are those where the lower and upper bounds are active given the signs of the vector ν.

$$\begin{array}{rcl}
\mathcal{L}_{act}(\nu) & = & \{k|\nu_k > 0, a_k = l_k\} \\
\mathcal{U}_{act}(\nu) & = & \{k|\nu_k < 0, a_k = u_k\} \\
\mathcal{F}_{act}(\nu) & = & \left\{j|\nu_j \neq 0\right\} \setminus (\mathcal{L}_{act}(\nu) \cup \mathcal{U}_{act}(\nu))
\end{array}$$

The indices in $\mathcal{F}_{act}(\nu)$ are those variables that are not active given ν. Thus, f is at its minimum when $\mathcal{F}_{act}(\tau_f) = \varnothing$. Alternatively, f is at a minimum when $\mathcal{U}_{act}$ and $\mathcal{L}_{act}$ include all of the non-basic variables on the row,

$$|\mathcal{U}_{act}(\tau_f)| + |\mathcal{L}_{act}(\tau_f)| + 1 = \|T_f\|_s .$$

We can then see that the inequality derived for constraining the minimization of (3.2) is also an instance of Lemma 2.21.

To decrease the value of a_f, we choose some x_j s.t. $j \in \mathcal{F}_{act}(\tau_f)$ and determine an appropriate δ for $\text{UPDATE}(j, \delta)$. (We discuss the strategy for picking x_j later in this section.) The direction in which we attempt to move a_j is determined by $T_{f,j}$: if $T_{f,j} < 0$, then we want $\delta \geqslant 0$ and if $T_{f,j} > 0$, then we want $\delta \leqslant 0$. Since the UPDATE operation must maintain the invariant $l \leqslant a \leqslant u$, the value of δ is

[2] The row vectors τ_i and $\text{mask}_{i,i}$ (defined in Sec.2.2.8) are equal by construction.

constrained by the bounds on x_j:[3]

$$l_j \leqslant a_j + \delta \leqslant u_j.$$

Also, for every basic variable x_i that depends on x_j, the value a_i must stay within bounds: $l_i \leqslant a_i + T_{i,j} \cdot \delta \leqslant u_i$. These cases can be unified using τ:

$$\text{for all } k, \quad l_k \leqslant a_k + \tau_{k,j}\delta \leqslant u_k.$$

PRIMAL always considers UPDATE(j, δ) operations that are maximal: the value of δ is selected so that at least one variable's assignment is pushed against its bound (any larger change would violate the bound). For each k, the candidate value for δ is the one that sets x_k equal to one of its bounds (which bound is determined by the sign of δ and the sign of $\tau_{k,j}$). We call these candidate values for δ the *break points* of x_j. Formally, let $\delta_U(j, k, \alpha)$ be the amount x_j must change in order to make x_k equal to α after an UPDATE:

$$\delta_U(j,k,\alpha) = \frac{\alpha - a_k}{\tau_{k,j}}, \text{and} \tag{3.3}$$

$$\delta_B(j,k) = \begin{cases} \delta_U(j,k,l_k) & T_{f,j} \cdot \tau_{k,j} > 0 \\ \delta_U(j,k,u_k) & T_{f,j} \cdot \tau_{k,j} < 0 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.4}$$

The break points for x_j are all defined values of $\delta_B(j, k)$.

In PRIMAL, for each j, we simply select k to minimize $|\delta_B(j, k)|$ (ties can be broken by picking the minimum k). The operation UPDATE$(j, \delta_B(j, k))$ then maintains the invariant that no variable violates its bound. Additionally, the assignment to x_k is guaranteed to be pressed up against its bound. When $j \neq k$, x_k is a basic variable, so we can allow for (potential) future progress by pivoting x_k out of the basis and replacing it with x_j, PIVOT(j, k).[4,5] The strategy of minimizing $|\delta_B(j, k)|$ to select the x_k to leave the basis is a leaving rule. By always selecting updates like this, PRIMAL ensures that $a(f)$ monotonically decreases.

We have just described a rule for selecting x_k given x_j, but we need the corresponding entering rule for selecting x_j. A simple way to ensure termination is to select the smallest index j in $\mathcal{F}_{act}(\tau_f)$. This is the Primal variant of Bland's

[3] This departs from classical presentations of primal simplex which assume the initial assignment a' assigns all non-basic variables to either their upper or lower bounds. This removes the need to consider this case.

[4] Note the change of the order of PIVOT and UPDATE from Chapter 2.

[5] The $j = k$ case handles the possibility of updating x_j to one of its bounds. Tracing through the definitions, we see that for $j \in \mathcal{N}$, $\tau_{j,j} = 1$, $\delta_U(j, j, \alpha) = \alpha - a_j$, and $\delta_B(j, j)$ is either $l_j - a_j$, $u_j - a_j$ or undefined.

rule. A better heuristic is to select x_j so as to maximize the value of $|T_{f,j}|$. This is often called Dantzig's rule. Dantzig's rule tends to be dominated in practice by more sophisticated rules, such as steepest-edge or devex, but these are out of the scope of this discussion [53,56,63,98]. The algorithm PRIMAL(f) in Figure 3.1 is a minimization routine that repeatedly selects an update and pivot until $\mathcal{F}_{act}(\tau_f)$ is empty and then returns the minimum value found for f. For the purposes of this thesis, we ignore unbounded problems, i.e. problems where a_f can take on arbitrarily low values [19, 56, 98]. (To handle this case, change the while loop condition additionally to stop once a_f is set to $-\infty$ assuming the intermediate operations support extended arithmetic.) The selection procedure uses a terminating variant of Dantzig's rule (it follows Dantzig's rule as long as $\delta_B(j,k)$ is nonzero, otherwise it follows Bland's rule). Note that when $\delta_B(j,k) \neq 0$, the value of f strictly decreases, which makes it impossible to return to any previous state (as all previous states had larger values of f). Thus, the presence of a minimization function makes it easier to rule out cycles (the source of nonterminating runs). Termination only needs to be addressed for cases when f gets stuck and stops decreasing.

3.2 Sum of Infeasibility

The primal simplex algorithm of the previous section finds an assignment that optimizes a linear function f given an initial feasible assignment a'. This is known as a *Phase II* simplex algorithm. A *Phase I* simplex algorithm either finds a feasible assignment to the input problem, $Ta = 0$ and $l \leqslant a \leqslant u$, or concludes that none exists. Broadly speaking, SIMPLEXFORSMT in Chapter 2 may be viewed as an example of a Phase I simplex algorithm. (It may be more correct to call SIMPLEXFORSMT and SOISIMPLEX "simplex-like algorithms" as the objective function changes every round.) A Phase I pass may also be done using the primal simplex algorithm on a transformed version of the input problem. This transformed problem is straightforward to initially satisfy, and its optimal assignments can be transformed into feasible assignments iff the initial problem is feasible [21, Section 8.4].

The SIMPLEXFORSMT routine focuses on searching for an assignment a that satisfies $l \leqslant a \leqslant u$. The set of error variables, E, are the variables that violate one of their bounds. Let V_i denote the amount by which x_i violates its bound:

$$V_i = \begin{cases} a_i - u_i & a_i > u_i \\ l_i - a_i & a_i < l_i \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

The violation function is formally a function over variables, $V(x_i)$, but is generally written as the vector that is the point-wise application of the function over $\mathcal{X}$, i.e. $V_i = V(x_i)$. By construction, V_i is nonnegative and piecewise linear, and x_i satisfies its bounds iff $V_i = 0$. Finding a satisfying assignment requires reducing each V_i to 0. The coefficient of a_i in V_i is formalized as the direction of violation λ_i. Locally, minimizing V_i is equivalent to minimizing $\lambda_i x_i$ where λ_i is -1 if $a_i > u_i$, 1 if $a_i < l_i$, and 0 otherwise.

$$\lambda_i = \begin{cases} +1 & a_i > u_i \\ -1 & a_i < l_i \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

Further, let $VC(k)$ be the constraint restricting the minimization of $\lambda_k x_k$.

$$VC(k) = \begin{cases} c : x_k \leqslant u_k & a_i > u_i \\ c : x_k \geqslant l_k & a_i < l_i \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.7}$$

Due to the invariant in SIMPLEXFORSMT that $Ta = 0$, whenever $E = \varnothing$ the assignment a was feasible. By the invariant that all non-basic variables are in bounds, E is a subset of the basic variable indices $\mathcal{B}$. Every round of SIMPLEXFORSMT can be seen as minimizing the function $\lambda_i \tau_i \cdot \mathcal{X}$. For every basic variable x_i in error, $i \in E$, the non-basic variables on its row (the variables with non-zero coefficients) can be partitioned into either non-basic flexible variables of row i that enable the function $\lambda_i x_i$ to decrease,

$$\mathcal{F}_{act}(\lambda_i \tau_i)$$

or those with constraints that are assigned to their corresponding bounds. The constraints for these non-basic variables are

$$\mathcal{L}_{act}(\lambda_i \tau_i) \qquad \text{and} \qquad \mathcal{U}_{act}(\lambda_i \tau_i)$$

Remark. If $\sigma \neq 0$ and $i \in \mathcal{B}$, then $|\mathcal{F}_{act}(\sigma\tau_i)| + |\mathcal{L}_{act}(\sigma\tau_i)| + |\mathcal{U}_{act}(\sigma\tau_i)| + 1 = \|T_i\|_s$.

Using the function V, we can construct the *sum-of-infeasibilities* function. For a given assignment, the sum of infeasibilities is given by:

$$V(\mathcal{X}) = \sum_{x_i \in \mathcal{X}} V_i \tag{3.8}$$

For a given assignment, $V(\mathcal{X})$ is a real value. Let $V_{\mathcal{X}}$ denote the result of re-

```
procedure SOICHECK
    while $\exists j \in \mathcal{N} \cap E$ do
        UPDATE$(j, -\lambda_j \cdot V_j)$
    CHECKBASICVARIABLESFORCONFLICTS (E)
    while $\mathcal{F}_{act}(\tau_f) \neq \varnothing \wedge$ Conflict $\notin$ outputStream do
        $\langle i, \delta, j \rangle \leftarrow$ SOISELECT()
        PIVOTANDUPDATE$(i, \delta, j)$
        CHECKBASICVARIABLESFORCONFLICTS ($\{k | T_{k,i} \neq 0\} \cap E$)
    if Conflict $\in$ outputStream then
        return Conflict
    else if $E = \varnothing$ then
        return (Sat $a$)
    else
        SoiQE() $\rightarrow$ outputStream (Sec. 3.3.5)
        return Conflict
```

Figure 3.3: Check procedure for SOISIMPLEX.

placing a_i by x_i in the definition of V. The evaluation of the polynomial $V_{\mathcal{X}}(x_i)$ by a given assignment a is equal to V_i, i.e. $a \circ V_{\mathcal{X}}(x_i) = V_i$. Similarly $V_{\mathcal{X}}(\mathcal{X})$ is a polynomial over $\mathcal{X}$ for a given assignment a such that $a \circ V_{\mathcal{X}}(\mathcal{X}) = V(\mathcal{X})$. Thus there are now three equivalent conditions for the assignment a satisfying $l \leqslant a \leqslant u$: $E = \varnothing$, $V(\mathcal{X})$ is the value 0, $V_{\mathcal{X}}(\mathcal{X})$ is the polynomial 0. Also, let VB_i denote the violated bound on x_i: either l_i, u_i or undefined. So whenever $V_i \neq 0$, then $V_i = \lambda_i(a_i - VB_i)$.

3.3 Sum of Infeasibilities Simplex

In this section, we introduce a Simplex-based theory solver for QF_LRA which we call SOISIMPLEX. The function minimized is the sum of infeasibilities of all of the variables, $V(\mathcal{X})$. Minimizing the sum of infeasibilities is a standard technique for finding an initially feasible assignment for linear programs [19, 56].

We assume the same setup as in the previous section: we start with a fixed (modulo pivoting) tableau and a satisfying assignment a. The SAT solver asserts a set of literals that determine the upper and lower bounds for the variables. The theory solver must provide a check routine that either reports satisfiable (with a satisfying assignment) or unsatisfiable (with a conflict). The main loop for SOISIMPLEX uses essentially the same machinery to minimize $V_{\mathcal{X}}(\mathcal{X})$ as was used in PRIMAL for minimizing a linear function f. However, there are a number

```
1: procedure SOISELECT
2:     $S \leftarrow \varnothing$
3:     for $j \in \mathcal{F}_{act}(\tau_f)$ do
4:         leaving $\leftarrow \varnothing$
5:         for all $k$ such that $k = j \vee T_{k,j} \neq 0$ do
6:             leaving $\leftarrow$ leaving $\cup \{\langle \delta_U(j, k, l_k), k \rangle\}$
7:             leaving $\leftarrow$ leaving $\cup \{\langle \delta_U(j, k, u_k), k \rangle\}$
8:         select $\langle \delta, k \rangle \in$ leaving to minimize $\langle \Delta V(j, \delta), |\delta|, k \rangle$
9:         $S \leftarrow S \cup \langle j, \delta, k \rangle$
10:    select $\langle j, \delta, k \rangle \in S$ minimizing $\left\langle \mathrm{sgn}(\Delta V(j, \delta)) \cdot |T_{f,j}|, j \right\rangle$
11:    return $\langle j, \delta, k \rangle$
```

Figure 3.4: Selection rules for SOISIMPLEX.

of complications caused by the fact that $V_{\mathcal{X}}(\mathcal{X})$ is only piecewise linear instead of linear. The majority of this section is devoted to handling these challenges.

Because we cannot represent the optimization function $V_{\mathcal{X}}(\mathcal{X})$ directly in the tableau, we use a linearized approximation. First note that that

$$V(\mathcal{X}) = \sum_{x_i \in \mathcal{X}} V_i = \sum_{x_i \in \mathcal{X}} \lambda_i \cdot (a_i - VB_i).$$

In some neighborhood of a_i, the value of $\lambda_i \cdot VB_i$ will be constant. Discarding this term and replacing a_i with x_i results in the polynomial $f(\mathcal{X}) = \sum_{x_i \in \mathcal{X}} \lambda_i \cdot x_i$. Note that the function still depends on the current assignment (which determines λ_i), but for a given assignment, the function is linear. Using the rows in the tableau T, we can substitute for the basic variables and rearrange the sums to get:

$$f = \sum_{j \in \mathcal{N}} \left(\sum_{x_i \in \mathcal{X}} \lambda_i \tau_{i,j} \right) \cdot x_j.$$

We use this polynomial in roughly the same way we used f in PRIMAL: it is the 0th variable and it is always basic. To compute the tableau row for f, we simply compute coefficients for each nonbasic variable x_j by adding, for each row i, the entry in column j multiplied by the directional multiplier λ_i. The computed coefficients depend on λ_i and thus have to be updated every time the assignment changes. This can be implemented efficiently by instrumenting UPDATE to detect when λ_i changes to λ_i' for some i. When this happens, we update f's row (T_f) as follows: $T_f \leftarrow T_f + (\lambda_i' - \lambda_i) \cdot \tau_i$.

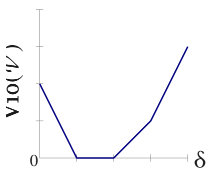
The check procedure for SOISIMPLEX is given in Fig. 3.3. It iterates while:

no row contains a conflict (**Conflict** $\notin$ `outputStream`), and there is an inactive non-basic variable on f's row ($\mathcal{F}_{act}(\tau_f) \neq \emptyset$). If some conflict has been reported, then SIMPLEXFORSMTCHECK safely terminates with the discovered conflict[s]. If $\mathcal{F}_{act}(\tau_f)$ and E are empty, the current assignment is satisfying. Otherwise, $E \neq \emptyset$, $\mathcal{F}_{act}(\tau_f) = \emptyset$, and f is at a minimum. Section 3.3.5 discusses extracting a conflict in the latter situation.

As in the PRIMAL algorithm, the selection procedure in Figure 3.4 iterates over all $j \in \mathcal{F}_{act}(\tau_f)$. The leaving rule considers x_j as well as every basic variable x_k where $T_{k,j}$ is nonzero. We consider two possible updates (break points) for each such variable: one which sets it to its upper bound and one which sets it to its lower bound. Unlike PRIMAL, we consider updates for which some new basic variable could become violated. However, we still ensure that global progress is made. We denote by $\Delta V(j, \delta)$ the amount that $V(\mathcal{X})$ would change if we were to change the current assignment by executing UPDATE(j, δ). From all of the possible leaving variables and updates, we then select the pair for which $\Delta V(j, \delta)$ is minimal (equivalently, the pair that reduces the value of $V(\mathcal{X})$ the most). Section 3.3.1 describes how to efficiently compute the values for $\Delta V(j, \delta)$. We also show in that section that for each x_j, there is always a choice of $\langle \delta, k \rangle$ such that $\Delta V(j, \delta) \leqslant 0$. This ensures that $V(\mathcal{X})$ monotonically decreases. Tie breaking for the leaving rule is done by selecting the minimum value of $|\delta|$ and then the minimum variable index k. The motivation for the former is discussed in subsection 3.3.2.

The entering rule selects between candidate triples $\langle j, \delta, k \rangle$ for $j \in \mathcal{F}_{act}(\tau_f)$. Any triple for which $\Delta V(j, \delta)$ is negative ensures that SOISIMPLEX is making progress. This allows for SOISIMPLEX to treat $V(\mathcal{X})$ in a manner analogous to a_f in PRIMAL. Following our modified Dantzig's rule, we select the entering variable with the largest coefficient so long as it decreases $V(\mathcal{X})$ with ties being broken by selecting the variable with the smaller index.

$$T: \begin{aligned} f &= -2 \cdot x_2 + x_3 \\ x_1 &= 2 \cdot x_2 - x_3 \end{aligned} \qquad \begin{aligned} 3 \leqslant\ & x_1 \leqslant 7 \\ & x_2 \leqslant 3 \\ 1 \leqslant\ & x_3 \end{aligned}$$



$$a(x_1) = 1 \qquad a(x_2) = 1 \qquad a(x_3) = 1$$

Figure 3.5: Simple example showing $V(\mathcal{X})$ after UPDATE(x_2, δ).

Example We show how SOISIMPLEX works using the simple example shown in Fig. 3.5. With the given assignment, the bound $x_1 \geqslant 3$ is violated, and $V(\mathcal{X}) =$

2. The variable x_2 is flexible, and we examine it for updates. The break points for x_2 are at $\delta \in \{1, 2, 3\}$, and correspond to changes to x_2 that respectively set x_1 to its lower bound, x_2 to its upper bound, and x_1 to its upper bound. Figure 3.5 shows how the value of $V(\mathcal{X})$ changes if x_2 is updated by δ. For $\delta \in \{1, 2\}$, $\Delta V(2, \delta) = -2$ and $V(\mathcal{X})$ will become 0. Because of the tie-break on $|\delta|$, the pair $\langle \delta.k \rangle = \langle 1, 1 \rangle$ is selected, and then the triple $\langle 2, 1, 1 \rangle$ is returned. After the call to UPDATE, the algorithm terminates with a satisfying solution.

3.3.1 Computing $\Delta V(j, \delta)$

To implement line 22 of SOISELECT, we must compute the values of $\Delta V(j, \delta)$ for every break point δ. We use the fact that the polynomial $V_{\mathcal{X}}$ is linear between break points and that the slopes of these linear segments can be computed. Let Δ be a increasing sorted list of the positive δ values in `leaving`, and let $\delta_0 = 0$: $0 = \delta_0 < \delta_1 < \ldots$. Let κ_i be the set of values of k that are paired with δ_i in `leaving`. We proceed as follows. We know that $\Delta V(j, 0) = 0$ and that the slope m_0 as δ increases from 0 is $T_{f,j}$. Now, we can compute:

$$\Delta V(j, \delta_i) = \Delta V(j, \delta_{i-1}) + m_{i-1} \cdot (\delta_i - \delta_{i-1}).$$

Furthermore, we know that at δ_i, each variable x_k (for $k \in \kappa_i$) transitions to satisfying its bound or violating its bound, meaning that λ_k will change at δ_i to some λ'_k. This change can be used to compute the slope m_i for the next segment:

$$m_i = m_{i-1} + \sum_{k \in \kappa_i} (\lambda'_k - \lambda_k) \cdot \tau_{k,j}.$$

Continuing this walk over increasing values of δ computes $\Delta V(j, \delta)$ for all $\delta \geqslant 0$. Another analogous pass can be done to compute the $\Delta V(j, \delta)$ values for negative δ values. A number of nice properties follow from the above computation, including the following lemma:

Lemma 3.1. *For each $j \in \mathcal{F}_{act}(\tau_f)$, there is some pair $\langle \delta, k \rangle \in$ `leaving` such that $\Delta V(j, \delta) \leqslant 0$.*

Proof. If $\delta = 0$ is a break point, then $\Delta V(j, 0) \leqslant 0$. Now assume 0 is not a break point. The x_j's considered are on f's row so $T_{f,j} \neq 0$. If $T_{f,j} > 0$, there must exist some $\lambda_i \cdot \tau_{i,j} > 0$. So there exists a negatively-valued break point, $\delta_U(j, i, VB(i))$. Let δ be the negative break point closest to 0. We know that $\Delta V(j, \delta) = 0 + T_{f,j} \cdot \delta < 0$. Similarly, if $T_{f,j} < 0$, then $\Delta V(j, \delta) < 0$ for the minimal positive δ. □

The proof further suggests that it is sufficient to consider either just the negative or just the positive values of δ (depending on the value of $T_{f,j}$) without affecting correctness.

3.3.2 Termination

The termination of SOISIMPLEX is again based on the termination of Bland's rule. Suppose that SOISIMPLEX does not terminate. There are only a finite number of possible assignments that can be considered as the number of variables is finite, and every change to the assignment assigns a variable x_j to either $u(x_j)$ or $l(x_j)$. Because the value of $V(\mathcal{X})$ is determined by the assignment and monotonically decreases, any nonterminating execution must have an infinite tail during which $V(\mathcal{X})$ is unchanged and the update selected, $\langle j, \delta, k\rangle$ is such that $\Delta V(j, \delta) = 0$. As was shown in the proof of Lemma 3.1, if the minimal $\Delta V(j, \delta)$ found is 0, then $\delta = 0$ must be a break point. The leaving rule enforces that the δ selected minimizes the tuple $\langle \Delta V(j, \delta), |\delta|, k\rangle$. So in the tail of a nonterminating execution $\Delta V(j, \delta) = 0$ and $\delta = 0$ at every step. Thus after this point, no variable is changing in assignment and no variable changes its relationship to its bounds. Every leaving and entering variable is then selected based on picking the minimum index. The argument that PRIMAL cannot cycle under Bland's rule can then be directly applied. We refer readers interested in the proof of the termination of Bland's rule to [56,98].

3.3.3 Heuristics and $V(\mathcal{X})$

Instead of examining all $j \in \mathcal{F}_{act}(\tau_f)$ for the best candidate, we can instead just look at heuristically many candidates. The search can stop once a candidate has been found that makes progress (i.e. $\Delta V(j, \delta) < 0$). Further, there is more freedom in selection heuristics than we have shown here. In particular, one can use any heuristic desired until no progress has been made for a while. CVC4's implementation for example uses a heuristic that prefers shorter columns until progress stalls and then uses Bland's rule.

3.3.4 Fast calculation of conflicts at break points

During the calculation of break points, it is possible to determine if pivoting x_j with x_i would result in a row conflict on x_j's new row in $O(1)$ time by using the $\mathtt{g}_{\pm 1,i}$ and $\mathtt{h}_{\pm 1,i}$ values. This is because the sign of $T_{i,j}$ is known during the calculation of the break points for x_j as well as whether or not x_j would become an error variable after this candidate pivot. Such selections are always preferred. CVC4's selection also heuristically prefers the set E to be as small as possible.

3.3.5 Conflicts with Multiple Rows

If the sum of infeasibilities function is provably at a minimum and no single row produces a conflict, but the assignment is not yet feasible, we can still detect a conflict and derive an explanation. When the non-basic variables on f's row are assigned to their upper or lower bounds (depending on the sign of their coefficient), we can conclude that f is minimized by the current assignment.[6] This is entailed by the non-basic variables on f's row being at their upper or lower bounds and the equalities in the tableau:

$$T\mathcal{X} = 0 \wedge \bigwedge_{\tau_{f,j}>0} x_j \geqslant l_j \wedge \bigwedge_{\tau_{f,k}<0} x_k \leqslant u_k \models_{\mathbb{R}} f \geqslant a_f \tag{3.9}$$

Replacing f by its definition, we get that $\sum_{i \in E} \lambda_i x_i \geqslant a_f$. We subtract from both sides of the inequality $\sum_{i \in E} \lambda_i \, VB_i$ to get:

$$\sum_{i \in E} \lambda_i (x_i - VB_i) \geqslant a_f - \sum_{i \in E} \lambda_i \, VB_i = \sum_{i \in E} \lambda_i (a_i - VB_i). \tag{3.10}$$

Note that in (3.10) the left hand side is now exactly $V_{\mathcal{X}}(\mathcal{X})$ and the right-hand side is exactly $V(\mathcal{X})$. We are in the case that the assignment is not feasible so the sum of infeasibilities function $V(\mathcal{X}) > 0$. We can now conclude the following entailment:

$$T\mathcal{X} = 0 \wedge \bigwedge_{\tau_{f,j}>0} x_j \geqslant l_j \wedge \bigwedge_{\tau_{f,k}<0} x_k \leqslant u_k \models_{\mathbb{R}} \sum_{i \in E} \lambda_i x_i > \sum_{i \in E} \lambda_i \, VB_i \tag{3.11}$$

For each of the violated bounds, we know that $\lambda_i x_i \leqslant \lambda_i \, VB_i$ is equivalent to either $x_i \geqslant l_i$ or $x_i \leqslant u_i$. Thus the sum of all of these constraints imply that $\sum_{i \in E} \lambda_i x_i \leqslant \sum_{i \in E} \lambda_i \, VB_i$. This is in conflict with the entailment in (3.11). This allows us to extract the following conflict:

$$\bigwedge_{\tau_{f,j}>0} x_j \geqslant l_j \wedge \bigwedge_{\tau_{f,k}<0} x_k \leqslant u_k \wedge \bigwedge_{i \in E} \lambda_i x_i \leqslant \lambda_i \, VB_i \wedge T\mathcal{X} = 0 \tag{3.12}$$

(Lemma 3.2 (given later in this Section) formalizes these conflicts.)

Explanations constructed like this may not be minimal. Consider taking any infeasible system like the one above $T\mathcal{X} = 0$, $l \leqslant \mathcal{X} \leqslant u$, and the assignment minimizes f (formally, τ_f). Using (3.12), we get some conflict C. Construct a duplicate problem and assignment $T'\mathcal{X}' = 0$, $l' \leqslant \mathcal{X}' \leqslant u'$ where all of the variables are primed. We know that this duplicate problem also has a conflict

[6] The is the condition $\mathcal{F}_{act}(\tau_f) = \varnothing$ on line 5 in Figure 3.3).

C'. Combine these two disjoint problems into a single problem:

$$[T; T'][\mathcal{X}; \mathcal{X}'] = 0, l \leqslant \mathcal{X} \leqslant u, l' \leqslant \mathcal{X}' \leqslant u'.$$

The combined assignment $[a; a']$ minimizes the sum-of-infeasibilities for the combined system $V(\mathcal{X} \cup \mathcal{X}')$. (Each assignment minimizes its part of the disjoint subproblems.) The conflict extracted using (3.12) is going to be exactly the union of the two conflicts $C \cup C'$; however, either C or C' would be better conflicts for the combined problem.

One could use the QuickXplain conflict minimization algorithm [71,72] starting from the conflict (3.12) to find a minimal conflict. This adaptive algorithm adds and removes constraints and performs additional consistency checks to derive minimal conflicts. Using the variant of QuickXplain in [72], minimizing the conflict may require up to $2k \cdot \log \frac{|E|}{k} + 2k$ additional consistency checks (where k is the number of basic variables in the minimal conflict). Each of these consistency checks corresponds to a new Simplex invocation.

The paper [73] mentions an adaptation of the QuickXplain algorithm for [heuristically] minimizing the conflict without additional Simplex search. The algorithm we present more closely follows the variant of QuickXplain from [71]. Intuitively, we are going to try to find a subset of the error variables E such that the sum of the infeasibilities of just this set of variables is minimized by the current assignment. If we can find such a set, we can extract a conflict following (3.12).

We generalize f to work over arbitrary sets of basic variables. Given a subset S of $\mathcal{B}$, we denote by f^S the row vector that is the sum of the [effective] rows for the indices in S multiplied by their violation direction.[7]

$$f^S = \sum_{i \in S} \lambda_i \tau_i \quad \text{and} \quad f^S \cdot \mathcal{X} = \sum_{j \in \mathcal{N}} \left(\sum_{i \in S} \lambda_i \cdot T_{i,j} \right) \cdot x_j.$$

We note that $f^E = f$ and whenever f^S is minimized, the preconditions to generating a conflict using 3.12 apply. (f^S is minimized when $\mathcal{F}_{act}(f^S) = \varnothing$.)

The algorithm in Figure 3.6 is essentially the same abstract algorithm as QuickXplain from [71] with a new conflict detection scheme. The procedure takes as input two sets of basic variable indices S_1 and S_2 such that we know a conflict like (3.12) is known to exist in their union $S_1 \cup S_2$. (Formally, the conditions for the conflict are that $V_i > 0$ for all $i \in S_1 \cup S_2$, $S_1, S_2 \subseteq \mathcal{B}$, and $\mathcal{F}_{act}(f^{S_1 \cup S_2}) = \varnothing$.) The set of indices S_1 are treated as fixed. The procedure attempts to find a minimal subset of indices X such that $S_1 \subseteq X \subseteq S_1 \cup S_2$, and

[7] Note that f^S is simply a row vector. It is not added to the tableau as f was.

```
Require: $\mathcal{F}_{act}(f^{S_1 \cup S_2}) = \varnothing$ and $\forall i \in S_1 \cup S_2 . V_i > 0$
 1: procedure SoiQuickXplain($S_1, S_2$)
 2:     if $S_1 \neq \varnothing$ then
 3:         $X \leftarrow S_1$
 4:     else
 5:         select an arbitrary $i$ $in S_2$
 6:         $X \leftarrow \{i\}$ // ensure X is not trivially empty
 7:     Let $f^X \leftarrow \sum_{i \in X} \lambda_i \tau_i$
 8:     while $\mathcal{F}_{act}(f^X) \neq \varnothing$ do
 9:         select some j such that $f_j^X \neq 0$ and $\mathrm{sgn}(f_j^X) \neq \mathrm{sgn}(f_j^{S_1 \cup S_2})$
10:         select some $i \in S_2 \setminus X$ such that $\mathrm{sgn}(\lambda_i T_{i,j}) = -\mathrm{sgn}(f_j^X)$.
11:         $i_{last} \leftarrow i$, $f^X \leftarrow f^X + \lambda_i \tau_i$, $X \leftarrow X \cup \{i\}$
12:     if $X = S_1$ then
13:         return X
14:     else
15:         $R \leftarrow C \cup \{i_{last}\}$
16:         enumerate the elements of $X \setminus R$ as $i_1, \ldots i_k$
17:         $X_1 \leftarrow \left\{ i_1, \ldots, i_{\lfloor k/2 \rfloor} \right\}$, $X_2 \leftarrow \left\{ i_{\lfloor k/2 \rfloor + 1}, \ldots, i_k \right\}$
18:         if $X_2 \neq \varnothing$ then
19:             $R_2 \leftarrow$ SoiQuickXplain($R \cup X_1, X_2$)
20:             $R \leftarrow R \cup (R_2 \setminus X_1)$
21:         if $X_1 \neq \varnothing$ then
22:             $R \leftarrow$ SoiQuickXplain($R, X_1$)
        return R
```

Figure 3.6: A heuristic conflict minimization algorithm for sum of infeasibility conflicts based on QuickXplain.

Lemma 3.2 applies to X. On the initial call, S_1 is empty and $S_2 = E$. The algorithm begins with $X = C$ and constructs the row vector f^X. The algorithm then keeps adding a new index i from S_2 to X until it is known that a conflict can be extracted from f^X. (We discuss selecting i in the next paragraph.) This process is guaranteed to terminate as $f^{S_1 \cup S_2}$ is a candidate solution, and X converges to $S_1 \cup S_2$. Hopefully, X is a strict subset of $S_1 \cup S_2$ and not all of the elements of S_2 have been added, but this is not guaranteed. With the knowledge that a conflict can be generated from X, we attempt to build a subset R of X such that $\mathcal{F}_{act}(f^R) = \varnothing$ holds. At this point, the last index added (i_{last}) and S_1 are heuristically assumed to be required ($R \leftarrow S_1 \cup \{i_{last}\}$). The algorithm then partitions the elements of X that are not in R in half as X_1 and X_2. The algorithm attempts to recursively minimize X_2 assuming $R \cup X_1$. The indices from X_2 included in the returned result are added to R. The algorithm then attempts to minimize X_1 assuming that the indices in R are included in the sum. The algorithm finally returns the indices in R and those selected from X_1.

Lines 8-11 in Figure 3.6 describe how row indices are selected to be added to X. The vector f^X is the sum of rows vectors $\lambda_i \cdot \tau_i$ for each $i \in X$. The loop keeps running while a conflict cannot be constructed for X i.e. while $\mathcal{F}_{act}(f^X)$ is non-empty. Because all of the non-basic x_js are at their bounds on the row $f^{S_1 \cup S_2}$, the only way $\mathcal{F}_{act}(f^X)$ can be non-empty is that f^X and $f^{S_1 \cup S_2}$ disagree on the sign of some non-basic variable x_j.

$$\operatorname{sgn}(f^X_j) \neq \operatorname{sgn}(f^{S_1 \cup S_2}_j)$$

The coefficient of this x_j must either cancel to zero or flip signs by adding more elements from S_2 into X. Thus there must be some $i \in S_2$ that has not yet been added to X such that $\operatorname{sgn}(\lambda_i T_{i,j}) = -\operatorname{sgn}(f^X_j)$. We add such an i to X and continue the loop until a conflict is found. This cancellation rule is main insight of this algorithm.

We now formalize the correctness of such sum-of-infeasibility conflict generation. For all $i \in \mathcal{B}$, let y_i be the row vector s.t. $T_i = y_i A$. (See Lemma 2.16 for more details on extracting y_i.) Let $z^S = \sum_{i \in S} \lambda_i y_i A$ for a subset S of $\mathcal{B}$. The k'th element of z^S is denoted z^S_k. The vector z^S differs from f^S because z^S also includes non-zero coefficients for basic variables.

$$f^S = z^S + \sum_{i \in S} \lambda_i e_i$$

The analogs of the sets $\mathfrak{L}$, $\mathfrak{U}$ and $\mathfrak{R}$ from Section 2.2.8 for a fixed set S are defined as:

$$\begin{aligned}
\mathfrak{L}^S &= \left\{ x_k \geqslant l_k \middle| z_k^S > 0 \right\}, \\
\mathfrak{U}^S &= \left\{ x_k \leqslant u_k \middle| z_k^S < 0 \right\}, \text{ and} \\
\mathfrak{R}^S &= \left\{ A_k \cdot \mathcal{X} = 0 \middle| z_k^S \neq 0, k \in \mathrm{Aux} \right\}.
\end{aligned}$$

Lemma 3.2. *If* $V_i > 0$ *for all* $i \in S$*, and* $\mathcal{F}_{act}(f^S) = \emptyset$*, then* $\mathfrak{L}^S \cup \mathfrak{U}^S \cup \mathfrak{R}^S \models_{\delta\mathbb{R}}$ **false**.

Proof. The selection of the sets $\mathcal{L}^S$ and $\mathcal{U}^S$ over z^S create a value γ for the sum of upper and lower bounds over z^S. (See Section 2.2.2 for details.)

$$\gamma = \sum_{i \in \mathcal{L}} z_i l_i + \sum_{j \in \mathcal{U}} z_j u_j$$

By case analysis, we can show that $\gamma = \sum_{i \in S} V_i$. Thus $\gamma > 0$. Corollary 2.14 is then immediately applicable to z^S, $\mathcal{L}^S$, $\mathcal{U}^S$, $\mathfrak{R}^S$ and γ. (The proof also works for comparisons over $\delta\mathbb{R}$.) See the Appendix Section A.2.7 for a proof that traces these connections more explicitly. □

The constraints that correspond to $\mathfrak{L}^S \cup \mathfrak{U}^S \cup \mathfrak{R}^S$ can then be returned as a $\mathcal{T}_{\mathbb{R}}$-conflict (as was done in Section 2.2.9). It is also possible to apply the conflict strengthening technique from Section 2.2.9 to the resulting conflict where $\sum_{i \in S} V_i$ is the initial surplus. This has not yet been implemented in CVC4.

A possible alternative algorithm to SoiQuickXplain is to try all subsets of E. While the worst case performance is exponential $\binom{|E|}{k}$, combining this with the cancellation rule used in the SoiQuickXplain (lines 8-11) makes this much more efficient in practice. For $k = 2$, this turns out to yield an efficient incomplete procedure for finding minimal conflicts.

Lemma 3.3. *If* $|S| = 2$ *and* $\mathcal{F}_{act}(f^S) = \emptyset$ *and* $\mathcal{F}_{act}(f^{\{i\}}) \neq \emptyset$ *for all* $i \in S$*, then* $\mathfrak{L}^S \cup \mathfrak{U}^S \cup \mathfrak{R}^S$ *is a minimal conflict.*

Proof. We give a sketch of the proof. Let i and j be an enumeration of S. Let k be an element in $\mathcal{F}_{act}(f^{\{i\}})$. As $k \notin \mathcal{F}_{act}(f^S)$, $T_{i,k} = -T_{j,k}$. Update the assignment to move the surplus of a_i onto a_k. Now all of the literals except for the bound on x_j have been satisfied. Starting from this new assignment an argument similar as the one in the proof of Lemma 2.23 shows that relaxing any of these constraints results in a satisfiable set of constraints. □

3.4 Experimental Results

In this section, we compare CVC4 against itself using two different sets of options.[8] The first set of options uses the default solver, an implementation of SIMPLEXFORSMT (which is a bit better than the version that won the `QF_UFLRA` division—which includes `QF_LRA`—of SMT-COMP 2012 [24]). The second set of options enables a new implementation of SOISIMPLEX. The two configurations of CVC4 are run with most other heuristics disabled so that the comparison is an accurate reflection of the performance of the two algorithms as described in this paper.[9] The comparison is done on the `QF_LRA` benchmarks from the `SMT-LIB` library [12] as well as a new family of benchmarks from biological modeling, `latendresse` [76]. The `latendresse` family of benchmarks is a set of problems that originated from an analysis of biochemical reactions using the flux-balance analysis method.[10] The `miplib` and `latendresse` families are of particular interest as they contain the only timeouts in these experiments. These problems are characterized by relatively little propositional structure, and a large and relatively dense input tableau. All of the experiments were conducted on a 2.66GHz Core2 Quad running Debian 7.0 with a time limit of 1000 seconds. Every example stays below a memory limit of 2GB. Overall, SOISIMPLEX solves 636 while SIMPLEXFORSMT solves only 629. Interestingly, SOISIMPLEX is slightly slower on the `SMT-LIB` benchmarks (see Fig. 3.7), and even solves one fewer benchmark (the satisfiable `miplib` benchmark `fixnet-7000.smt2`), but solves all of the `latendresse` benchmarks while SIMPLEXFORSMT times out on 8 of them.

To understand these results better, we recorded how many pivots were done (for both algorithms) during each call to the respective check routines for benchmarks that both algorithms are able to solve. For the `SMT-LIB` benchmarks, almost all queries sent to the theory solver are "easy" for the simplex solvers (both SIMPLEXFORSMT and SOISIMPLEX). Table 3.1 shows, for given numbers of pivots (or ranges of numbers of pivots), the number of calls to check whose pivot count is in that range. These numbers are expressed as:

- $D(n)$ is the number of calls to SIMPLEXFORSMTCHECK with n pivots in the experiment.
- $\sum D(n)$ is the total number of pivots performed by the $D(n)$ calls.

[8] Experiments were run using the submission to SMT-EVAL 2013: CVC4 version 1.2, available at `github.com/CVC4/CVC4/tree/smteval2013`.

[9] Both solvers are run with `--new-prop --no-restrict-pivots`. SOISIMPLEX is run with the additional flag `--use-soi`. The `--no-restrict-pivots` flag disables stopping simplex after K pivots at non-leaf SIMPLEXFORSMTCHECK calls ($K = 200$ by default).

[10] When the experiment was originally run in 2013, the `latendresse` benchmarks were not a part of `SMT-LIB`. They have since been accepted for inclusion into `SMT-LIB`'s `QF_LRA` family.

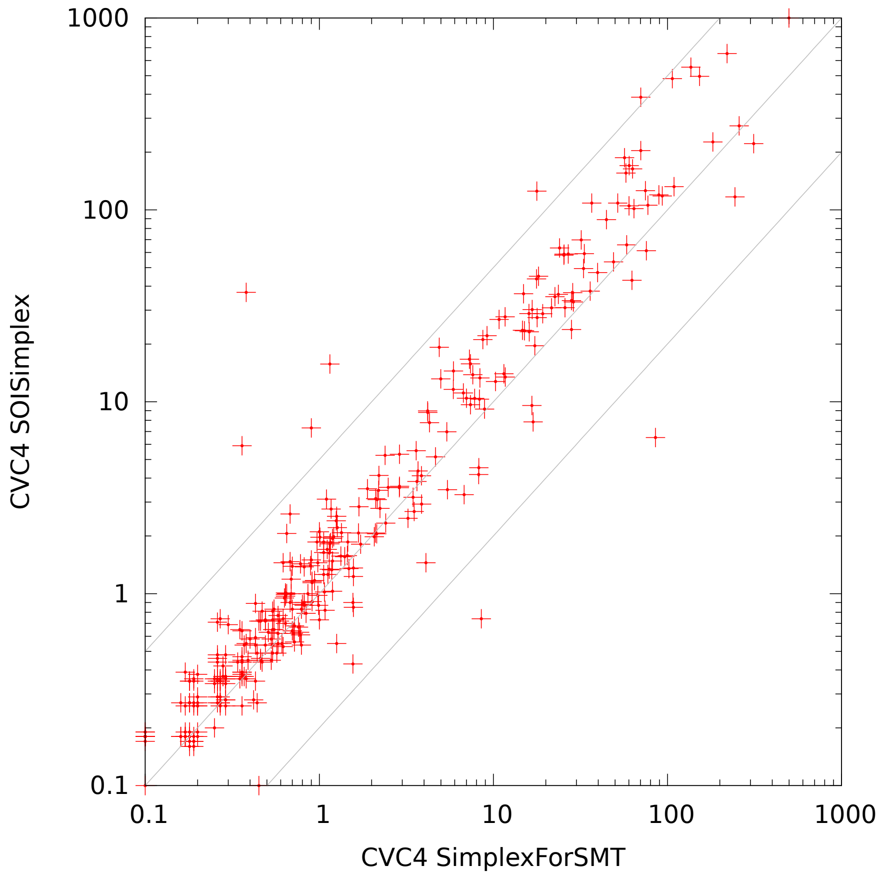


Figure 3.7: Log-scaled running times (sec.) for experiment 1 on the `QF_LRA` benchmarks from `SMT-LIB`.

Range	0	1	[2, 10]	[11, 100]	[101, 1000]	[1000, 2238]	total
$D(n)$	32832**k**	645**k**	896**k**	174**k**	2362	7	34551**k**
$\sum D(n)$	0	645**k**	3677**k**	3628**k**	479**k**	10**k**	8440**k**
$V(n)$	30475**k**	924**k**	1008**k**	130**k**	655	0	32539**k**
$\sum V(n)$	0	924**k**	3900**k**	2366**k**	126**k**	0	7317**k**

Table 3.1: Number of pivots per call to check for experiment 1. (See the text for a description of $D(n)$ and $V(n)$). **k** is an abbreviation for 1000.

- $V(n)$ is the number of calls to SOICheck with n pivots.
- $\sum V(n)$ is the total number of pivots performed by the SOICheck with n pivots.

The maximum number of pivots for any single call to check is 2238. The number of pivots is generally very low and on average, SOISimplex uses fewer pivots than SimplexForSMT.

The 8 timeouts by SimplexForSMT on `latendresse` have a very different signature. Each of them times out in the middle of a very long SimplexForSMTCheck call performing thousands of pivots. On average, the interrupted SimplexForSMTCheck routines had performed 18263 pivots and had been running 937s [/1000s]. This first experiment confirms our expectation that SOISimplex is effective at reducing the number of pivots required to solve challenging instances.

3.4.1 Take Aways

The author believes these experiments demonstrate both the strength and weakness of SimplexForSMT's local optimization criteria. It is good at keeping the amount of work small in the context of a DPLL($\mathcal{T}$) style search. The local optimization criteria requires little analysis and is quite an efficient heuristic for many SMT problems; however, its global convergence is questionable on large and hard examples. SOISimplex adds a global optimization criterion and appears to be more robust for large and hard examples, but this comes with the cost of additional analysis during pivot selection.

Future work will explore how to heuristically take advantage of the best characteristics of both algorithms.

Chapter 4

Extending Simplex to Integer and Integer Real Arithmetic

Chapters 2 and 3 describe decision procedures for solving QF_LRA using variants of the Simplex algorithm. This chapter discusses established techniques for extending these theory solvers to additionally handle the theories of integer arithmetic and integer real arithmetic. Quantifier-Free Linear Integer Arithmetic (QF_LIA) may be intuitively thought of as QF_LRA where all of the variables are restricted to take on integer values. For example, if x and y are variables with the sort Real, the literals

$$2 < 2y + 2x, 1 \geqslant 2y - 2x, 4x - 2 \leqslant y \tag{4.1}$$

are satisfied by the assignment $\left[x \to 0, y \to \frac{1}{2}\right]$. If instead x and y had the sort Int, this assignment does not satisfy the literals as y is not assigned to a value in $\mathbb{Z}$. As we will show later, this example is unsatisfiable if x and y are restricted to take integer values.

Integer arithmetic over the symbols $\langle 0, 1, +, \cdot, < \rangle$ is often referred to as elementary arithmetic. We follow SMT terminology and call this the theory of integers $\mathcal{T}_{\mathbb{Z}}$. The restriction that variables only take on integer values makes the decision problem for $\mathcal{T}_{\mathbb{Z}}$ significantly more challenging than $\mathcal{T}_{\mathbb{R}}$. This is generally true for fragments with and without quantifiers. Kurt Gödel famously showed in 1931 that if $\mathcal{T}$ is a consistent theory and $\mathcal{T}'$ is a recursively enumerable subtheory of $\mathcal{T}$ that is capable of expressing a limited subset of the axioms of elementary arithmetic, then for any deduction system there must exist a sentence ϕ such that $\phi \in \mathcal{T}$ cannot be deduced from $\mathcal{T}'$ [57].[1] This implies that any such theory $\mathcal{T}$ is undecidable. This is quite different than the situation for the theory $\mathcal{T}_{\mathbb{R}}$ which admits quantifier elimination and is thus complete and decid-

[1] See [49] for more details.

able. Gödel's result ensures that elementary arithmetic is undecidable. Further, Matiyasevich showed that Hilbert's 10th problem, the satisfiability of quantifier-free non-linear integer arithmetic is undecidable [82]. Luckily, the logic of linear integer arithmetic admits quantifier elimination and is thus complete and decidable [49,64]. Most efforts of the SMT community have focused on linear integer arithmetic, and in particular the `QF_LIA` logic.

The common scheme for building a theory solver for `QF_LIA` builds upon a Simplex based theory solver for `QF_LRA`. The literals for `QF_LIA` are subjected to additional simplifications to strengthen the constraint (Section 4.3). The Simplex solver determines if the constraints $\mathcal{A}$ are unsatisfiable in $\mathcal{T}_{\mathbb{R}}$. If the input constraints $\mathcal{A}$ evaluate to **true** under the $\mathcal{T}_{\mathbb{R}}$ assignment a, and a happens to assign all of the variables to $\mathbb{Z}$ values, then a is also a $\mathcal{T}_{\mathbb{Z}}$ assignment (Section 4.4). If a is not an integer assignment, techniques such as cutting planes (Section 4.4.2) and branching (Section 4.4.1) are used to successively refine the assignment.

4.1 Quantifier-Free Linear Integer Arithmetic

In integer arithmetic, the set of variables $\mathcal{X}$ contains only variable symbols labeled with the sort `Int`, $\{x_1 : \texttt{Int}, x_2 : \texttt{Int}, \dots, x_n : \texttt{Int}\}$. Like $\Sigma_{\mathbb{R}}$, the core signature $\Sigma_{\mathbb{Z}}$ is $\langle 0, 1, +, \cdot, < \rangle$; however, the functions of $\Sigma_{\mathbb{Z}}$ are over the sort `Int` instead of `Real`. The definition is extended to include all integer constants, and the comparison operations $\langle >, \leqslant, \geqslant \rangle$. Linear terms for integers are defined in a similar manner to the real case. For the linear terms s and t, the atoms of Quantifier-free Linear Integer Arithmetic (`QF_LIA`) are of the form $s \bowtie t$ where $\bowtie \in \{=, <, >, \leqslant, \geqslant\}$.

This thesis follows the SMT-LIB semantics of the theory of integer $\mathcal{T}_{\mathbb{Z}}$. The SMT-LIB v2.0 standard defines $\mathcal{T}_{\mathbb{Z}}$ using a standard model where `Int` is mapped into $\mathbb{Z}$, the function symbol $+$ is mapped to the mathematical function $+$ over the integers, etc. [107]. This assumes the existence of the standard model.

The linear fragment may alternatively be axiomatized by Presburger arithmetic [49]. Presburger arithmetic admits quantifier elimination and is thus complete and decidable [34]. Fischer and Rabin showed that deciding satisfiability of linear integer arithmetic has a strict $2^{2^{cn}}$ non-deterministic-time lower bound (for some $c > 0$) where n is the size of the formula [52]. Oppen gave a $2^{2^{2^{cn}}}$ deterministic time upper bound (for some $c > 0$) for the same problem [92].

Christos Papadimitriou gave a proof that if there exists a solution to an Integer Programming problem then there exists a solution a such that each component of a is $\leqslant n(mq)^{2m+1}$ where n is the number of variables, m is the number of rows, and q is the absolute value of the largest integer appearing in the input problem [93]. The number of bits required to represent such a solution is polynomial in the number of input bits of the problem. (This result assumes the variables are restricted to be natural numbers.) This result extends to show `QF_LIA` is in the complexity class **NP**. In principle, `QF_LIA` solvers can be made complete by precomputing bounds for all of the variables using this method. Due to the size of these bounds and the complexity of integrating this technique within the context of DPLL($\mathcal{T}$), this has not yet been attempted in the literature.

4.2 Quantifier-Free Linear Integer Real Arithmetic

Linear integer arithmetic is extended to linear integer real arithmetic. We denote the quantifier-free fragment as `QF_LIRA`. Some complications are introduced into this language by the requirements of many-sorted logic. The variables of `QF_LIRA` are partitioned into a set of variables of sort `Int` and a set of variables of sort `Real`. The signature $\Sigma_{\mathbb{Z}\mathbb{R}}$ has two copies of the standard symbols $\langle 0, 1, +, \cdot, < \rangle$. One copy of these standard symbols operates over terms of sort `Real` and another operates over terms of sort `Int`. The signature $\Sigma_{\mathbb{Z}\mathbb{R}}$ also contains conversion functions to and from `Int` and `Real`: `ToInt` and `ToReal`. The conversion function `ToReal` maps a term of the sort `Int` to the sort `Real`, and `ToInt` maps `Real` to `Int`. The function `ToReal` is in a sense trivial as it is required to just transform an integer value into the same integer value in the domain. The reverse conversion function `ToInt` maps a real value to the greatest integer value less than or equal to the real value (the floor function). For compactness the signature is extended by the integer constants (with integer sort), and rational constants (with the real sort). The definition of linear terms follows that of linear real arithmetic and linear integer arithmetic.

The theory of integer real arithmetic $\mathcal{T}_{\mathbb{Z}\mathbb{R}}$ is defined using a standard model. Entailment and satisfaction in this theory is written as $\models_{\mathbb{Z}\mathbb{R}}$. Weispfenning gave an algorithm for quantifier elimination of linear integer real arithmetic [110].[2] Thus this logic is complete. We can again safely restrict our interest to the natural domains of $\mathbb{R}$ for `Real` and $\mathbb{Z}$ for `Int`.

4.3 Normalization and Simplification

Normalization for linear terms in `QF_LIRA` and `QF_LIA` is roughly the same as `QF_LRA` with two significant extensions. `QF_LIA` is handled as a subcase of `QF_LIRA`. The first extension is to normalize with two sort symbols `Int` and `Real`.

[2] The proof requires the `ToInt` function to be a part of the language. Weispfenning also defines `Int` to be a subsort of `Real`.

The second is strengthening inequalities using the knowledge that variables are integer.

Alternations of sort symbols present a challenge to rewriting. As it is simpler to deal with only terms of one sort, the normalization of QF_LIRA terms begins by rewriting all of the terms of sort Int into terms of the sort Real. The one exception is that an integer variable x can appear directly under the function ToReal in the form (ToReal x). The first step in this transformation is to remove (ToInt t) for a term t : Real by replacing the (ToInt t) term by a new skolem variable x : Int, and restricting x by:

$$(\texttt{ToReal}\ x) \leqslant t \wedge t < (\texttt{ToReal}\ x) + 1.$$

Applications of integer comparison functions ($=_{\texttt{Int}}$, $<_{\texttt{Int}}$, etc.) are converted to their real sort equivalents by wrapping the right and left-hand sides by ToReal, e.g. $s =_{\texttt{Int}} t$ becomes (ToReal s) $=_{\texttt{Real}}$ (ToReal t). The function ToReal is now distributed across sums and multiplication by constants:

$$(\texttt{ToReal}(+_{\texttt{Int}}\ s\ t)) \xrightarrow{\text{Rewriter}} (+_{\texttt{Real}}\ (\texttt{ToReal}\ s)\ (\texttt{ToReal}\ t)).$$

The integer constants wrapped by ToReal are converted to the corresponding rational constant. Exhaustive application of these rules results in a formula where the only terms with sort Int are integer variables wrapped by ToReal. After this point, we simply write x_i instead of (ToReal x_i) for the appearance of an integer variable. As a simplification, we additionally assume that the indices for all integer variables are larger than indices for all real variables, e.g. if x : Real and y : Int, then $x \prec y$. After normalizing the sorts, normalization for linear terms then proceeds the same way to produce terms of the form $c_0 + \sum_{i=1}^{N} c_i x_i$ for rational constants $c_0, c_1, \ldots, c_N$.

Normalization for atoms in QF_LIRA proceeds roughly in the same fashion as the real case. Atoms of the form $(\bowtie\ s\ t)$ for $\bowtie \in \{=, \leqslant, <, >, \geqslant\}$ are handled by first rewriting the term $s - t$ to the form $c_0 + \sum c_i x_i$. This is converted to $\sum c_i x_i \bowtie -c_0$. If the left-hand side is empty (contains no variables), the comparison $0 \bowtie -c_0$ is evaluated to either **true** or **false**. Assuming the left-hand side contains at least one variable and at least one variable has sort Real, the normalization proceeds as it did for QF_LRA.[3] If all of the variables in $\sum c_i x_i$ are integer, then additional simplification may be done to strengthen the literals.

For example, the atom $2y - 2x \leqslant 1$ from (4.1) may be rewritten to first $\frac{1}{2} \geqslant y - x$. If x : Int and y : Int, the left hand side may be strengthened to $\lfloor \frac{1}{2} \rfloor = 0$.

[3] This check may be done in $O(1)$ time by looking at the first variable as all real variables precede integers variables.

By perform such simplifications, the constraints in (4.1) can be normalized to

$$x + y \geqslant 1, x - y \geqslant 0, 4x - y \leqslant 2. \tag{4.2}$$

Because of tightening these constraints, the previous assignment $\left[x \to 0, y \to \frac{1}{2}\right]$ no longer satisfies the first constraint (interpreting x and y as real variables). The assignment $\left[x \to \frac{1}{2}, y \to \frac{1}{2}\right]$ does satisfy these tighter constraints, but again does not assign both x and y to integer values.

Formally, normalization for an integer linear relation $\sum_1^N c_i x_i \bowtie d$ begins by multiplying by the least common multiple of the denominators in the rational constants $d, c_1, \ldots, c_N$ to get integer values $d', c'_1, \ldots, c'_N$. Strict inequalities are transformed to non-strict inequalities by adding or subtracting one to the right-hand side.

$$\sum c'_i x_i < d' \xrightarrow{\text{Rewriter}} \sum c'_i x_i \leqslant d' - 1$$
$$\sum c'_i x_i > d' \xrightarrow{\text{Rewriter}} \sum c'_i x_i \geqslant d' + 1$$

The atoms for inequalities involving only integer variables must therefore use $\bowtie \in \{=, \leqslant, \geqslant\}$. Let d'' be this new right-hand side. Next, normalization computes the greatest common denominator g of $c'_1, \ldots, c'_N$. If g is not a divisor of d'' ($g \nmid d''$), then the relation can be tightened based on $\bowtie$.

- If $\bowtie$ is "$=$", then this is rewritten to **false** as it is equivalent to a sum of integer terms being equal to a non-integer rational d''.
- If $\bowtie$ is "$\leqslant$", then the atom is rewritten to $\sum \frac{c'_i}{g} x_i \leqslant \left\lfloor \frac{d''}{g} \right\rfloor$.
- If $\bowtie$ is "$\geqslant$", then the atom is rewritten to $\sum \frac{c'_i}{g} x_i \geqslant \left\lceil \frac{d''}{g} \right\rceil$.

If g is a divisor of d'', then the left and right-hand sides of the atom are simply divided by g. To maximize sharing, the constraint is multiplied by the sign of the leading left hand coefficient (potentially reversing inequalities). And to further increase sharing for inequalities, if the right hand side is odd, the relation is rewritten so that the right-hand side is even.

$$\sum c_i x_i \leqslant d \text{ and } 2 \nmid d \xrightarrow{\text{Rewriter}} \neg\left(\sum c_i x_i \geqslant d + 1\right)$$
$$\sum c_i x_i \geqslant d \text{ and } 2 \nmid d \xrightarrow{\text{Rewriter}} \neg\left(\sum c_i x_i \leqslant d - 1\right)$$

Preprocessing for theory solvers directly follows `QF_LRA` (Section 2.1.2). Auxiliary variable introduction follows the scheme discussed in 2.2.3.[4] When a negated inequality on an integer variable comes into the theory solver, it will be of the form $\neg(x_i \leqslant z)$ or $\neg(x_i \geqslant z)$ (where z is an integer constant) These literals are equivalent to $x_i > z$ and $x_i < z$. These strict inequalities are sent to the AssertLower or AssertUpper procedures which are extended to additionally tighten the inequalities to $x_i \geqslant z+1$ or $x_i \leqslant z-1$. This simplification ensures that the upper and lower bounds of all integer variables are always integer values.

4.4 Theory Solvers for `QF_LIRA`

We describe a theory solver based on the simplex theory solvers discussed in Chapters 2 and 3. We again assume that an assignment a maps each $x_i \in \mathcal{X}$ to a $\mathbb{R}$ value.[5] An assignment a satisfies an atom $\sum c_i x_i \bowtie d$ whenever $\sum c_i a_i \bowtie d$ holds, and a_i is in the set $\mathbb{Z}$ for all variables such that x_i : `Int`. We assume that the indices of the variables in $\mathcal{X}$ are partitioned into a set of indices $\mathcal{X}_\mathbb{R}$ for real variables and a set of indices $\mathcal{X}_\mathbb{Z}$ for integer variables. The *real relaxation* of a formula Φ is obtained by replacing each integer variable x in the formula with a fresh real variable x_{Relax} whenever x appears. The real relaxation of a formula Φ will be denoted Φ_{Relax}. An assignment is *integer-compatible* if for each $x \in \mathcal{X}_\mathbb{Z}$ the assignment for $a(x_{\mathsf{Relax}}) \in \mathbb{Z}$. If a formula's real relaxation is satisfied by some assignment, we say it is *real-feasible* (or just *feasible*). With some redundancy, a formula is *integer-feasible* if it is satisfiable (and *integer-infeasible* otherwise). Instead of introducing the x_{Relax} variables, the theory solver simply uses x. It ignores sort mismatches internally.

4.4.1 Branching

The current best implementations of theory solvers for mixed linear integer and real arithmetic use a sound but incomplete procedure that layers integer reasoning on top of a simplex-based theory solver for linear real arithmetic [61]. Given a set of assertion literals $\mathcal{A}$, the simplex-based theory solver is first used to solve $\mathcal{A}_{\mathsf{Relax}}$. The result of the procedure is either a conflict set or an assignment a (when $\mathcal{A}$ is real-feasible). In the first case, no additional work is necessary as a conflict set for $\mathcal{A}_{\mathsf{Relax}}$ is also a conflict set for $\mathcal{A}$. In the second case, the assignment a is examined to see whether it is integer-compatible. If not, more work is needed to refine the assignment. A technique known as *branching* generates the

[4]If all of the variables in the sum $\sum c_i x_i$ are integer, the auxiliary variable may also be marked as an integer variable.

[5] Internally, the variables are still assigned $\delta\mathbb{R}$ values.

lemma that any term of integer sort is either below the floor of $\alpha \in \mathbb{R}$ or above the ceiling of α.

$$\frac{t : \mathtt{Int} \qquad \alpha \in \mathbb{R}}{t \leqslant \lfloor \alpha \rfloor \vee t \geqslant \lceil \alpha \rceil}\ \textsc{Branch} \tag{4.3}$$

A simple branching technique to refine an integer-incompatible assignment is to select a variable $x_i \in \mathcal{X}_{\mathbb{Z}}$ whose assignment is non-integer, and then to branch x_i on the value a_i. The SAT solver will ensure that one of the two new bounds on x_i is asserted before reinvoking the theory solver. The assignment of x_i must change in order to satisfy this stronger linear relaxation.

CVC4 implements two branching modes. The dominant branching heuristic CVC4 employs is the Cuts From Proofs technique [44]. When successful, this algorithm generates a plane $\sum_{j=1}^{K} c_j x_j = c_0$ such that $c_0, c_1, \ldots, c_K \in \mathbb{Z}$, the greatest common divisor (gcd) g of $c_1, \ldots, c_k$ does not divide c_0, and $x_1 : \mathtt{Int}, \ldots x_k : \mathtt{Int}$. Thus $\sum c_j x_j = c_0$ is unsatisfiable in $\mathcal{T}_{\mathbb{Z}}$. The theory solver can then add the branch:[6]

$$\sum_{j=1}^{K} c_j x_j \leqslant c_0 \quad \vee \quad \sum_{j=1}^{K} c_j x_j \geqslant c_0.$$

The rewriting techniques for integer inequalities given earlier (Section 4.3) must be able to strengthen both branches:

$$\sum_{j=1}^{K} \frac{c_j}{g} x_j \leqslant \left\lfloor \frac{c_0}{g} \right\rfloor \quad \vee \quad \sum_{j=1}^{K} \frac{c_j}{g} x_j \geqslant \left\lceil \frac{c_0}{g} \right\rceil.$$

The implementation of these branches closely follows [61] with heuristics to control the growth of intermediate numerical values. If this fails to produce a branch, the solver falls back on a naive round-robin scheme to determine branches on structural variables. Additionally, the solver periodically interleaves round-robin branching instead of always using branches from the Cuts From Proofs module.

Successful Example Returning to the example from (4.1) and (4.2), the solver can branch on either x or y with the assignment $\left[x \to \frac{1}{2}, y \to \frac{1}{2}\right]$. Suppose that the solver adds the branch $x \leqslant 0 \vee x \geqslant 1$. The linear relaxation of both of these

[6] There are proposals to turn the planes generated during Cuts From Proofs into cuts [67, personal communication].

cases is unsatisfiable, and hence the original formula is unsatisfiable.

$$x + y \geqslant 1, x - y \geqslant 0, 4x - y \leqslant 2, x \leqslant 0 \models_{\mathbb{R}} \textbf{false} \tag{4.4}$$
$$x + y \geqslant 1, x - y \geqslant 0, 4x - y \leqslant 2, x \geqslant 1 \models_{\mathbb{R}} \textbf{false} \tag{4.5}$$

A similar analysis on the branch $y \leqslant 0 \vee y \geqslant 1$ would also show that the input is unsatisfiable. Given the conflict clauses for (4.4) and (4.5) and $x \leqslant 0 \vee x \geqslant 1$, it is possible to learn a clause that does not contain either $x \leqslant 0$ or $x \geqslant 1$ by resolution (Sec. 1.4).

$$\neg(x + y \geqslant 1) \vee \neg(x - y \geqslant 0) \vee \neg(4x - y \leqslant 2)$$

Diverging Example While branches are quite simple to implement and can often work in finding solutions, naive use of this heuristic often triggers an infinite sequence of branches. For example, branching is capable of diverging on the following inequality.

$$-20x_0 + 2x_1 + 3x_2 \geqslant 1 \tag{4.6}$$

Suppose we start with the initial assignment where all variables are assigned 0. The 0th iteration of simplex assigns x_0 to $-\frac{1}{20}$ and terminates. Suppose the branch $x_0 \geqslant 0$ is then chosen. In response to the branch literal, simplex assigns x_0 to 0 and x_1 to $\frac{1}{2}$ in the 1st iteration, and x_1 is eligible for branching. If the up branch continues to be chosen, the sequence of assignment in the equation below are selected and the solver never terminates.

Iteration	a_0	a_1	a_2
0	$-\frac{1}{20}$	0	0
1	0	$\frac{1}{2}$	0
2	$\frac{1}{20}$	1	0
3	1	$\frac{21}{2}$	0
4	$\frac{21}{20}$	11	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2k+1$	k	$10k + \frac{1}{2}$	0
$2k+2$	$k + \frac{1}{20}$	$10k+1$	0

If infinite sequences of branches such as this are not ruled out, then the solver may not terminate. As most SMT solvers do not rule out such cycles, they are not, strictly speaking, decision procedures.

4.4.2 Cutting Planes

Branching attempts to refine integer-incompatible solutions by splitting the search space into two pieces and exploring each piece separately. A complementary technique known as *cutting planes* attempts to remove integer-incompatible solutions without introducing case splits [88, 98]. Consider a set of assertions $\mathcal{A}$. A cutting plane is a plane through the solution space of the real relaxation of $\mathcal{A}$ that *cuts off* some of the integer-incompatible assignments. More precisely, $\sum c_i x_i = d$ is a cutting plane for $\mathcal{A}$ and $\mathfrak{h} \equiv \sum c_i x_i \leqslant d$ is a *cut* iff the following conditions hold: (i) every assignment satisfying $\mathcal{A}$ also satisfies $\mathfrak{h}$; and (ii) at least one assignment satisfying the real relaxation of $\mathcal{A}$ also satisfies $\neg\mathfrak{h}$.[7] Condition (ii) may be formally stated as:

$$\mathcal{A} \models_{\mathbb{Z}\mathbb{R}} \mathfrak{h} \quad \text{and} \quad \mathcal{A}_{\mathsf{Relax}} \not\models_{\mathbb{R}} \mathfrak{h}_{\mathsf{Relax}}.$$

The inequality $\mathfrak{h}$ can be safely added to $\mathcal{A}$ without changing any of the (integer-compatible) satisfying assignments. Note that a cut is always entailed by the integer-tightening of $\mathcal{A}$ and never entailed by the real relaxation of $\mathcal{A}$. Cuts can be implemented using theory lemmas, by sending the lemma $\mathcal{A} \Rightarrow \mathfrak{h}$ to the SAT solver. Previous work has looked at using Gomory and Mixed Gomory cuts in SMT solvers [47].

Gomory showed that a combination of cuts and solving linear systems with a simplex solver is a complete technique for guiding the solver [58,59] However, because of the resulting complexity of the derived cuts, cuts in practice tend to be generated judiciously and in combination with other techniques. CVC4's arithmetic solver currently has very limited support for Mixed Gomory Cuts and an implementation of Mixed Knapsack and complemented-Mixed Integer Rounding cuts [47, 79]. It does not have its own heuristics for generating cuts and is only enabled when guided by the techniques in Chapter 5.

4.4.3 Gomory Cuts

To give the reader a feel for the algorithms involved in cutting planes, we give a brief introduction to Mixed Gomory Cuts. This introduction follows [47]. Suppose that the basic variable x_i is an integer variable, the assignment to x_i is not integer compatible ($a_i \notin \mathbb{Z}$), and all of the non-basic variables on the row are equal to either their lower or upper bounds. The i'th row of the tableau ensures x_i is equal to a sum of the non-basic variables ($T_i \cdot \mathcal{X} = 0$). The assignment to x_i is determined by the assignment to the non-basic variables on the row ($T_i \cdot a = 0$).

[7] Often, an additional requirement is that $\mathfrak{h}$ is not satisfied by the current assignment a. We will not require this here.

Let us partition the indices of the non-basic variables on the row into those equal to their upper bounds or those equal to their lower bounds.

$$J = \{j | j \neq i, T_{i,j} \neq 0, a_j = l_j\}$$
$$K = \{k | k \neq i, T_{i,k} \neq 0, a_k = u_k\}$$

(For simplicity, assume $J \cap K = \varnothing$.) We denote the fractional part of a_i as f_0 ($f_0 = a_i - \lfloor a_i \rfloor$). Below in (4.7), we have taken the difference of the two equalities $x_i = \sum_{j \in \mathcal{N}} T_{i,j} x_j$and $a_i = \sum_{j \in \mathcal{N}} T_{i,j} a_j$.

$$x_i - a_i = \sum_{j \in \mathcal{N}} T_{i,j}(x_j - a_j) \tag{4.7}$$
$$x_i - \lfloor a_i \rfloor = f_0 + \sum_{j \in J} T_{i,j}(x_j - l_j) + \sum_{k \in K} T_{i,k}(x_k - u_k) \tag{4.8}$$

In (4.8), this difference has been rewritten using J, K and f_0. As $x_i - \lfloor a_i \rfloor$ is entailed to take on integer values, the right-hand side of (4.8) must also have to take on integer values. We next split J and K based on the sign of $T_{i,j}$.

$$J^+ = \{j | j \neq i, T_{i,j} > 0, a_j = l_j\} \qquad J^- = \{j | j \neq i, T_{i,j} < 0, a_j = l_j\}$$
$$K^+ = \{k | k \neq i, T_{i,k} > 0, a_k = u_k\} \qquad K^- = \{k | k \neq i, T_{i,k} < 0, a_k = u_k\}$$

The Gomory cut for this row will be

$$\sum_{j \in J^+} \frac{T_{i,j}}{1 - f_0}(x_j - l_j) - \sum_{j \in J^-} \frac{T_{i,j}}{f_0}(x_j - l_j)$$
$$- \sum_{k \in K^+} \frac{T_{i,k}}{f_0}(x_k - u_k) + \sum_{k \in K^-} \frac{T_{i,k}}{1 - f_0}(x_k - u_k) \geqslant 1. \tag{4.9}$$

Lemma 4.1. *The inequality (4.9) is entailed in $\mathcal{T}_{\mathbb{Z}\mathbb{R}}$ assuming $x_i \in \mathcal{X}_{\mathbb{Z}}$, $a(x_i) \notin \mathbb{Z}$ and $\{\kappa | T_{i,\kappa} \neq 0\} = \{i\} \cup J \cup K$.*

Proof. We can determine the signs of all of the terms in (4.8).

$$j \in J^+ \implies T_{i,j}(x_j - l_j) \geqslant 0 \qquad j \in J^- \implies T_{i,j}(x_j - l_j) \leqslant 0$$
$$k \in K^+ \implies T_{i,j}(x_j - u_j) \leqslant 0 \qquad k \in K^- \implies T_{i,j}(x_j - u_j) \geqslant 0$$

This means that we can immediately derive the following two inequalities:

$$-\sum_{j\in J^-}\frac{T_{i,j}}{f_0}(x_j-l_j)-\sum_{k\in K^+}\frac{T_{i,k}}{f_0}(x_k-u_k)\geqslant 0 \tag{4.10}$$

$$\sum_{j\in J^+}\frac{T_{i,j}}{1-f_0}(x_j-l_j)+\sum_{k\in K^-}\frac{T_{i,k}}{1-f_0}(x_k-u_k)\geqslant 0 \tag{4.11}$$

Consider the sum:

$$\sum_{j\in J}T_{i,j}(x_j-l_j)+\sum_{k\in K}T_{i,k}(x_k-u_k). \tag{4.12}$$

Note that $f_0+(4.12)$ is the right hand side of (4.8) and must be integer. To show that (4.9) is entailed, we case split on whether $(4.12)>0$ or $(4.12)\leqslant 0$. In both cases, we use the fact that $f_0+(4.12)$ is integer to strengthen either the inequality (4.11) or the inequality (4.10).

- Suppose that $\sum_{j\in J}T_{i,j}(x_j-l_j)+\sum_{k\in K}T_{i,k}(x_k-u_k)>0$. Add f_0 to both sides of the inequality to get $f_0+\sum_{j\in J}T_{i,j}(x_j-l_j)+\sum_{k\in K}T_{i,k}(x_k-u_k)>f_0$. As the left hand side must be an integer, we can round the right hand side f_0 to the closest integer ($\lceil f_0\rceil=1$).

  $$f_0+\sum_{j\in J}T_{i,j}(x_j-l_j)+\sum_{k\in K}T_{i,k}(x_k-u_k)\geqslant 1$$

  We drop from the left hand side all of the negative terms ($j\in J^-$ and $k\in K^+$), and subtract f_0 from both sides of the inequality:

  $$\sum_{j\in J^+}T_{i,j}(x_j-l_j)+\sum_{k\in K^-}T_{i,k}(x_k-u_k)\geqslant 1-f_0.$$

  Dividing by $1-f_0$ (which is >0) yields:

  $$\sum_{j\in J^+}\frac{T_{i,j}}{1-f_0}(x_j-l_j)+\sum_{k\in K^-}\frac{T_{i,k}}{1-f_0}(x_k-u_k)\geqslant 1. \tag{4.13}$$

  Adding (4.13) and (4.10) yields (4.9).

- Suppose that $\sum_{j\in J}T_{i,j}(x_j-l_j)+\sum_{k\in K}T_{i,k}(x_k-u_k)\leqslant 0$. We again add f_0 to both sides of the inequality. The resulting left-hand side will be an integer, and we can take the floor of the right hand side to get:

$$f_0 + \sum_{j \in J} T_{i,j}(x_j - l_j) + \sum_{k \in K} T_{i,k}(x_k - u_k) \leqslant \lfloor f_0 \rfloor .$$

We then add $-f_0$ to both sides. As $\lfloor f_0 \rfloor = 0$, the resulting right-hand side is $-f_0$. Next, we divide by $-f_0$, and drop the negative terms in the left hand side (now $j \in J^+$ and $k \in K^-$) to get:

$$-\sum_{j \in J^-} \frac{T_{i,j}}{f_0}(x_j - l_j) + \sum_{k \in K^+} \frac{T_{i,k}}{f_0}(x_k - u_k) \geqslant 1 \tag{4.14}$$

Adding (4.14) to (4.11) yields (4.9).

□

The literals that were used in the derivation of the cuts are

1. the row is equal to 0 ($T_i \cdot \mathcal{X} = 0$),[8]
2. x_i is an integer variable ($x_i \in \mathcal{X}_{\mathbb{Z}}$),
3. $l_j \leqslant x_j$ for all $j \in J$, and
4. $x_k \leqslant u_k$ for all $k \in K$.

The derivation we have gone through has only assumed that x_i is integer. It has not assumed that the non-basic variables are integer variables. The inequality in (4.9) is a Mixed Gomory cut. Because we are not taking advantage of some non-basic variables potentially being integer variables, the cut in (4.9) is not the strongest possible Mixed Gomory cut.[9] After deriving a cut it is usual to replace auxiliary variables with their definitions, and to rewrite the derived inequality using the normalization techniques given in Section 4.3.

Example To compute the Gomory cuts of the example (4.1), we need to know the state of the simplex solver at the time it reports that the assignment is real feasible. For the example, three auxiliary variables are added.

$$\begin{aligned} s_1 &= x + y & \qquad s_1 &\geqslant 1 \\ s_2 &= x - y & \qquad s_2 &\geqslant 0 \\ s_3 &= 4x - y & \qquad s_3 &\geqslant 2 \end{aligned}$$

[8] See Cor.2.17 for the literals that explain $T_i \cdot \mathcal{X} = 0$.

[9] See [47] for a derivation of a stronger cutting plane.

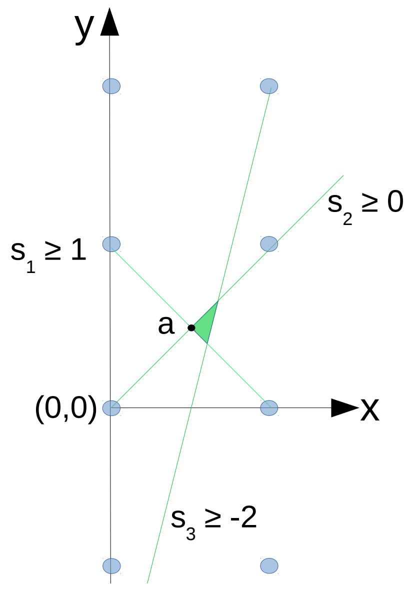


Figure 4.1: Geometric view of example (4.15). The intersection of the three half planes forms real-feasible space with the point a (the assignment) being a vertex of the space. The integer points of x and y are superimposed.

Beginning from the assignment of all variables to 0, one execution of simplex for these constraints is to first perform $\text{PIVOT}(y, s_1)$ and update s_1 to 1, and then to $\text{PIVOT}(x, s_2)$ and update s_2 to 0. This results in an assignment:

$$a_x = a_y = \frac{1}{2}, a_{s_1} = 1, a_{s_2} = 0, a_{s_3} = \frac{3}{2}$$

with the non-basic variables being s_1 and s_2. Figure 4.1 visually shows this example. The real-feasible assignment a is a vertex at the intersection of the two half planes $x + y \geqslant 1$ and $x - y \geqslant 0$. The tableau at this point is

$$T = \begin{array}{ccccc} s_1 & s_2 & s_3 & x & y \\ \hline 1/2 & -1/2 & & & -1 \\ 1/2 & 1/2 & & -1 & \\ 3/2 & 5/2 & -1 & & \end{array} \tag{4.15}$$

The non-basic variables s_1 and s_2 are equal to their lower bounds. Without going through the derivation, a Gomory cut can be derived from the second row in (4.15), and the facts that $s_1 \geqslant 1$ and $s_2 \geqslant 0$. The cut is

$$\frac{\frac{1}{2}}{1-\frac{1}{2}}(s_1 - 1) + \frac{\frac{1}{2}}{1-\frac{1}{2}}(s_2 - 0) \geqslant 1 \qquad \text{or} \qquad s_1 + s_2 \geqslant 2.$$

After removing auxiliary variables and simplification, this is equivalent to $x \geqslant 1$. After adding $x \geqslant 1$ as a constraint, the real relaxation is infeasible. Cuts may also be derived on the first and third rows. After simplification, the cut for the first row is coincidentally $x \geqslant 1$, and for the third row is $4x - y \geqslant 2$.[10] To understand why the first row also derives $x \geqslant 1$, note that y is not minimized by the assignment a while x and s_3 are minimized.

4.4.4 Mixed Integer Programming Solvers

The primal Simplex algorithm (Section 3.1) finds an assignment a that minimizes a linear function $f = \sum_{x_k \in \mathcal{X}} c_k x_k$ that satisfies a set of linear equalities $Ta = 0$ and the bounds $l \leqslant a \leqslant u$. The algorithm is seeded by an initial feasible assignment. If an initial feasible assignment is not provided, it is possible to employ techniques like those discussed in Chapters 2 and 3 to find an initial feasible assignment or to conclude that none exists. LP solvers based on simplex are often organized into two solving phases: Phase I finds an initial assignment, and Phase II finds an optimal assignment if Phase I finds an initial solution. It is also possible for the solver to conclude that the system has no finite minimum, i.e. it can take on any arbitrarily small $\mathbb{R}$ value.[11] Thus LP solvers usually return one of three results: the input problem is infeasible, an optimum assignment to f, or that the function f is unbounded.

The *Integer Programming* (IP) problem additionally restricts the set of candidate assignments of all variables to values over $\mathbb{Z}$. The *Mixed Integer Programming* (MIP) problem generalizes both the LP problem and IP problem by restricting only a subset of the structural variables to take integer values. We will only consider one popular architecture for MIP solvers, *branch-and-cut* MIP solvers. Branch-and-cut MIP solvers use roughly the same set of tools for solving optimization problems as theory solvers use for feasibility. A branch-and-cut execution is organized as a tree of related problems. Initially, the tree is the initial problem. If the real relaxation of the root problem is infeasible, then the problem has no solutions. If the problem is feasible, an optimal assignment of the

[10] Deriving a cut on row three requires s_3 to be labeled with `Int` during construction.

[11] In the formulation in Sec. 3.1, this occurs whenever a non-basic variable x_i is being examined and all of the break points are $-\infty$ or $+\infty$ Essentially, no bound for x_i's column restricts changing the value of x_i to decrease f. The code in 3.1 may easily be modified to handle this case.

real relaxation is found.[12] If that solution is integer compatible, then the solver returns it as the optimum value. If it is not integer compatible, then the solver may heuristically derive cuts. Generally, the derived cuts attempt to cut off the current integer incompatible assignment. The LP solver is then reinvoked to see if this new system is feasible, and if so, an optimal assignment is found, etc. This repeats until the solver is unable [or more likely heuristically unwilling] to generate new cuts. The solver then branches on an integer [structural] variable x_i whose current assignment is non-integer. This creates two subproblems: one in which $x \leqslant \lfloor a_i \rfloor$ and one in which $x \geqslant \lceil a_i \rceil$. The MIP solver then recursively solves both problem instances. If both are feasible, then the optimal answer of the current node is the larger of the two children. If one branch is infeasible then the optimal answer of the other child is the optimal of the parent. The third case is that both branches are infeasible and the current node is infeasible.

It is not necessary to fully optimize all branches. We discuss a style of early pruning that is often referred in the literature as *branch-and-bound*. The solver keeps around the best integer compatible assignment found so far, a_{Best}. The value of f for this assignment provides an upper bound on the global minimum value f can take. Once a node finds an optimal solution a_{Br} to the real relaxation, the value of f under this assignment is compared to the value of f on a_{Best}. If it is greater than or equal the best found so far, the branch is pruned as no integer compatible assignments can be less than a_{Br}. The assignment a_{Best} is updated whenever a branch that has not been pruned finds an optimal assignment that is integer compatible. As a convention, MIP solvers use the optimization function $f = 0$ to solve pure feasibility queries. This is a so called "bingo" mode as once any integer compatible solution is found, the solver can stop examining all other branches. If the solver is able to prune all branches without finding any integer compatible assignment, a_{Best} will not have been set in the course of execution and the solver concludes that the root node is infeasible.

[12] We are going to ignore unbounded IP and MIP problems in this discussion. Interested readers may consult [84].

Chapter 5

Leveraging Linear Programming Solvers

Because of their historical use in verification and theorem proving, SMT solvers typically use exact precision numeric representations internally in order to ensure that their calculations are correct and do not compromise the soundness of the overall system. For many typical SMT problems with significant Boolean structure (such as the majority found in the SMT-LIB benchmark library), this approach is sufficient, as the required theory reasoning is not too complex and the numbers involved in the internal calculations tend to stay relatively small. Moreover, such problems require tens or hundreds of thousands of calls to the theory solver. Thus, the theory solver's abilities to incorporate new constraints quickly, to rapidly detect inconsistencies, to propagate entailed literals, and to backtrack efficiently are far more important for overall efficiency than is the speed of the internal numerical calculations. However, there do exist problems for which this is not the case. If the internal simplex solver receives constraints that lead to large and dense linear systems, then using exact precision for the calculations required for the simplex search can overwhelm the solver.

Simplex-based linear programming (LP) solvers differ from SMT solvers in several important ways, including the following: (i) LP solvers solve only conjunctions of constraints - they cannot handle arbitrary Boolean combinations; (ii) LP solvers focus on both feasibility and optimization rather than just feasibility; (iii) LP solvers (generally) use floating point rather than exact precision arithmetic internally; and (iv) the product of many decades of research, modern LP solvers incorporate highly sophisticated techniques, making them very efficient in practice. The techniques used in LP solvers have been extended to the problem of optimizing constraints where all or some of the variables are required to be integers (Integer Programming (IP) and Mixed Integer Programming (MIP)).

On challenging simplex instances, LP and MIP solvers are considerably more efficient than the techniques used inside of SMT solvers. However, LP and MIP solvers are not optimized for rapid incremental calls which make them inefficient as theory solvers for many SMT applications. In addition, their use of floating point means that occasionally they will return a result that is not logically sound. In this chapter, we show how LP and MIP solvers can be efficiently and soundly incorporated into a modern SMT solver. Our work builds on similar previous efforts but is the first to succeed in obtaining a significant improvement over state-of-the-art SMT solvers.

The rest of this chapter assumes familiarity with the core concepts and data-structures of Simplex discussed in Section 2.1 and MIP branch-and-cut solvers discussed in Chapter 4. (It also selectively refers to discussion in Section 2.2.) Section 5.1 discusses our approach for integrating an LP solver in a theory solver for linear real arithmetic, and section 5.2 shows how to extend this strategy to use an MIP solver in a theory solver for linear integer arithmetic. Section 5.3 reports and discusses experimental results as well the history of CVC4 in the SMT-COMP. This chapter concludes with a discussion of future work in Section 5.4. The content of this chapter is to appear in FMCAD'14 [74].

5.1 Leveraging LP Solvers

The first contribution of this chapter is a method for leveraging the strengths of both SMT and LP solvers to construct an efficient and robust theory solver for linear real arithmetic. This idea has been explored before. Early work by Yu and Malik [111] reports results on using an LP solver as a theory solver for SMT, but the issue of potentially incorrect results from the LP solver is not addressed. Faure et al. [51] integrate several LP solvers into the Barcelogic SMT solver [18]. They use an exact solver to lazily check the results from the LP solver to ensure soundness. Finally, in recent work by de Oliveira and Monniaux [42], extensive experiments are done using an LP solver within OpenSMT [23]. In this work, the LP solver is called first and the results are used to "seed" the search in the exact solver. Thus most of the search is done by the LP solver, while the exact solver still ensures correctness.

In each of these studies, experimental results on SMT-LIB benchmarks show that existing SMT solvers outperform the experimental solvers modified to use LP solvers, even if the LP solver results are not checked for correctness. The main reason for this is that for these benchmarks (and the applications they represent), solving requires many related calls to the theory solver, each of which is relatively simple. The algorithms used in SMT solvers are optimized for this case and thus perform better, even though they use exact arithmetic which in

```
1: procedure BalancedSolve
2:     $c \leftarrow$ ExactSolve($k_{EX}$)
3:     if $c$ is Sat or Unsat then return $c$
4:     Construct $\widetilde{T}, \widetilde{l}, \widetilde{u}$ from $T, l, u$
5:     $\left\langle \widetilde{c}, \widetilde{a}, \widetilde{\mathcal{B}} \right\rangle \leftarrow$ LPSolve($k_{LP}, \widetilde{T}, \widetilde{l}, \widetilde{u}$)
6:     if $\widetilde{c}$ is Sat or Unsat then
7:         $a' \leftarrow$ ImportAssignment($\widetilde{a}$)
8:         $c \leftarrow$ ExactReseed($a', \widetilde{\mathcal{B}}$)
9:         if $c$ is Sat or Unsat then return $c$
10:    return ExactSolve($k_{FI}$)
```

Figure 5.1: The BalancedSolve procedure for linear real arithmetic.

general is much slower than floating point arithmetic. A solution to this problem advocated in [51] is to build a floating-point LP solver optimized for many, simple, related calls.

Here, we present an alternative approach. The idea is to take the two existing algorithms as they are and use each one only in cases where it is likely to do well. We thus use an exact solver optimized for fast incremental checks as the primary theory solver. However, we also instrument this solver so that it can detect when it is starting to have difficulty, and in these cases we have it call the LP solver.

5.1.1 The BalancedSolve Algorithm

The overall approach is given by the algorithm BalancedSolve shown in Figure 5.1. First, an efficient incremental exact solver ExactSolve is called with a heuristic cap on the number of pivots it may perform, k_{EX}. We assume that ExactSolve returns a status c (**Sat**, **Unsat**, or **Unknown**). If the exact solver returns **Sat** or **Unsat**, we are done and return the result. Otherwise, the heuristic cap was exceeded. In this case, the LP solver is called. We must convert the simplex problem described by T, l, and u to an analogous problem for the LP solver. We denote the LP analogs of the exact data by using the ~ annotation. They are constructed (following [42]) as follows. For each auxiliary variable x_s, the equality $A\,\mathcal{X} = 0$ ($x_s = \sum A_{s,j}\, x_j$), is added to $\widetilde{T}$ as

$$\widetilde{s} = \sum \text{float}(A_{s,j}) \cdot \widetilde{x_j},$$

where the conversion function float maps a rational to the nearest float. For each variable $\widetilde{x}$, the bounds $\widetilde{l}(x)$ and $\widetilde{u}(x)$ are constructed from the $\delta\mathbb{Q}$, $l(x)$ and $u(x)$

by approximating δ as a small constant ϵ. For example, if $l(x) = \langle c, d \rangle$, then $\widetilde{l}(x)$ becomes $\text{float}(c + \epsilon \cdot d)$.

The LP solver is invoked with its own pivot limit k_{LP}. If the LP solver terminates with **Sat** or **Unsat**, we retrieve the assignment $\widetilde{a}$ as well as the final set of basic variables $\widetilde{\mathcal{B}}$ from the LP solver. The LP assignment $\widetilde{a}$ is converted into a rational assignment a' by the IMPORTASSIGNMENT routine (described below). The EXACTRESEED procedure takes $\widetilde{\mathcal{B}}$ and a' and tries to verify the result of the LP solver using the exact solver. If this fails (or if the LP solver reaches its heuristic limit), the exact precision solver is run with a final limit k_{FI}. To ensure soundness, k_{FI} should be $+\infty$ for full effort calls to BALANCEDSOLVE i.e. calls made when there are no more decisions that can be made by the SAT engine in DPLL($\mathcal{T}$), but it can be heuristically less for non-final calls (Section 1.5.4).

We recall from Section 2.1 the core architecture of the Simplex search. Simplex solvers modify the assignment a and pivot the tableau T until a optimal assignment is found. An optimal assignment in a Phase II Simplex (optimization) search is a feasible assignment such that the assignment is provably at a maximum. For a Phase I or feasibility search, the optimal assignment is either any feasible assignment or an assignment and a tableau such that under the current assignment a variable (or set of variables) is optimal and its bound thus cannot be satisfied. The pieces of state that matter for deducing the optimality are the variable assignment a and the tableau T; however, the set of basic variable indices $\mathcal{B}$ induces a tableau (see Section 2.2.7).

5.1.2 Importing Assignments from the LP Solver

An important contribution to effectively using floating point solutions is the IMPORTASSIGNMENT procedure shown in Figure 5.2. A naive approach for converting a floating point assignment into a rational assignment would be to convert each floating point number to its exact rational equivalent. This has a number of drawbacks. The floating point numbers effectively contain a small amount of noise in the lower bits. This noise means that the most precise rational for this floating point is quite complicated when compared to the value of the calculation if it had been done in full precision.

As the assignment must satisfy $Ta = 0$ once it has been brought into the exact precision model, small amounts of noise make this unlikely to be satisfied. This may be partially fixed by changing the assignments to just the non-basic variables and letting these determine the assignment of the basic variables. This also requires care as Simplex tries to move towards (and between) points that are on the outside of the polytope of feasible solutions. If the assignment of a basic variable x_i is pushed to one of its bounds (say u_i), then small amounts of noise may cause the basic variable to violate its bound, $\sum c_j a_j = a_i \approx u_i$.

```
1: procedure ImportAssignment($\tilde{a}$)
2:     for all $x_i \in \mathcal{X}$ do
3:         $r \leftarrow$ DioApprox($\tilde{a}_i, D$)
4:         if $|r - a_i| \leqslant \epsilon$ then $r \leftarrow a_i$
5:         if $i \in \mathcal{X}_{\mathbb{Z}}$ and $|r - \lfloor r \rceil| < \epsilon$ then $r \leftarrow \lfloor r \rceil$
6:         if $r > u_i$ or $|r - u_i| \leqslant \epsilon$ then $r \leftarrow u_i$
7:         else if $r < l_i$ or $|r - l_i| \leqslant \epsilon$ then $r \leftarrow l_i$
8:         $a'_i \leftarrow r$
9:     return $a'$
```

Figure 5.2: The ImportAssignment procedure.

Intuitively, the ImportAssignment procedure attempts to assign each variable to a value that is close to the one given by the LP solver, but biased towards values that are easy to represent, partly because that makes them easier to calculate with, but also partly because the discarded portion often corresponds exactly to an accumulation of rounding error. For each variable x in the assignment, ImportAssignment first approximates $\tilde{a}_i$ as a rational using a technique based on continued fraction expansion called Diophantine approximation [88]. This technique finds the closest rational value with a denominator less than some fixed constant integer D. Next, we check to see if this value is within ϵ of the last known assignment for x_i in the exact solver. If so, the last known assignment is used. Next, if $i \in \mathcal{X}_{\mathbb{Z}}$ and the value is within ϵ of an integer z ($\lfloor r \rceil$ denotes the nearest integer to r), then z is used (note that this step only applies for mixed integer problems–see Section 5.2). Finally, ImportAssignment examines the value with respect to l_i and u_i. If the value violates one of these bounds or is within ϵ of a bound, then the bound is used instead.

5.1.3 Verifying the Output of the LP Solver

The ExactReseed routine, described in Fig. 5.3, attempts to duplicate the results from the LP solver within the exact solver. First the procedure updates the exact solver assignment by calling Update on each non-basic variable. Next it computes the set, Δ, of variable indices for variables that are non-basic in the exact solver but were marked as basic by the LP solver. We loop until as many variables in Δ as possible have been pivoted to become basic. At the beginning of each iteration, we visit all the rows of T to check for conflicts. (See Section 2.2.9 for a discussion of how to do this efficiently.) While checking for conflicts, we can also quickly detect whether any basic variable violates its upper or lower bound. If not, we have a satisfying assignment and can stop early. If neither

```
1:  procedure ExactReseed($a'$, $\widetilde{\mathcal{B}}$)
2:      for all $j \in \mathcal{N}$ do
3:          Update($j, a'_j - a_j$)
4:      $\Delta \leftarrow \mathcal{N} \cap \widetilde{\mathcal{B}}$
5:      while $\Delta \neq \varnothing$ do
6:          if $T$ contains a row conflict then
7:              return Unsat
8:          else if all variables satisfy their bounds then
9:              return Sat
10:         if $\exists\, i\, j.\, i \in \Delta \wedge j \notin \widetilde{\mathcal{B}} \wedge T_{i,j} \neq 0$ then
11:             Pivot($i, j$)
12:             Update($i, a'_i$)
13:             $\Delta \leftarrow \Delta \setminus \{i\}$
14:         else return Unknown
15:     return Unknown
```

Figure 5.3: The ExactReseed procedure.

check applies, we search for a pair of variables x_i and x_j such that i is in Δ meaning x_i is non-basic but should be basic, and $T_{i,j} \neq 0$ and $j \notin \widetilde{\mathcal{B}}$ meaning that x_j is basic but should be non-basic. If we can find such a pair, we pivot i and j and update the assignment of x_i to a'_i. Because of approximations made by the LP solver or by ImportAssignment, ExactReseed may fail to detect a satisfying assignment or a conflict in which case it returns **Unknown**. The ExactReseed procedure can be seen as using rounds of the simplex algorithm in [46] to achieve the same effect as ForcedPivot in [42,85].

An alternative to verifying the LP solution would be to use an exact external LP solver (e.g. [3,33,89]). However, the use of an exact external solver (as well as an attempt to implement their rather sophisticated techniques) is beyond the scope of this work.[1] Our goal, rather, is to make a first effort at an efficient integration of *inexact* floating-point solvers within SMT search. Integrating an exact external solver would be an interesting direction for future work. We discuss this idea in more detail in the future work section (Section 5.4).

Example To illustrate its importance, we consider an example where ImportAssignment makes a difference to ExactReseed. The example comes from the SMT-LIB `QF_LRA` problem, `pp08a-4000.smt2` in the `miplib` family. We exam-

[1] The ExactReseed does bear a strong resemblance to how these solvers restart the exact precision solver; however, it is unclear what the exact method of restarting the exact precision solvers these methods use which makes comparisons challenging.

ine a single row on which the variable tmp68 is basic.

$$\begin{aligned}\texttt{tmp68} =& 500 \cdot \texttt{pb\_x229} + 500 \cdot \texttt{pb\_x230} + 500 \cdot \texttt{pb\_x231} + \\ & 500 \cdot \texttt{pb\_x232} + 300 \cdot \texttt{pb\_x233} + 300 \cdot \texttt{pb\_x234}\end{aligned}$$

(The variables pb_x* are introduced as pseudo-Boolean variables for the Boolean variables x*.) The approximate assignment $\widetilde{a}(x_i)$ coming from the GLPK LP solver is given below for these variables as well an approximate decimal value:

x_i	$\widetilde{a}(x_i)$	$\approx \widetilde{a}(x_i)$
tmp68	2127816788229727/5026338869834	423.33333333333337
pb_x229	272945431961849/3275345183542189	0.08333333333333331
pb_x230	1/4	0.25
pb_x231	2573485501354571/15440913008127429	0.1666666666666666
pb_x232	857828500451524/5146971002709143	0.16666666666666669
pb_x233	1/10	0.1
pb_x234	1/5	0.2

These values do not satisfy the row by

$$3903637542271439/49388984424485044665355279878 \approx 7.9e - 14.$$

This is between 2^{-43} and 2^{-44}. Due to problems like this, the ExactReseed routine does not succeed if $\widetilde{a}$ is used directly. Using a', the assignment from ImportAssignment, the ExactReseed routine does succeed a real-feasible assignment a. Below is the assignment a' for the variables on the row for tmp68.

x_i	$a'(x_i)$
tmp68	1270/3
pb_x229	1/12
pb_x230	1/4
pb_x231	1/6
pb_x232	1/6
pb_x233	1/10
pb_x234	1/5

.

5.2 Using MIP Solvers to Improve Theory Solvers for Mixed Linear Integer and Real Arithmetic

In this section, we show how to extend the technique from the previous section to mixed linear integer and real arithmetic. Figure 5.4 shows the algorithm

```
1:  procedure IntegerSolve
2:      $c \leftarrow$ BalancedSolve()
3:      if $c$ is Unsat then return $c$
4:      Construct $\widetilde{T}, \widetilde{l}, \widetilde{u}$ from $T, l, u$
5:      $\left\langle \widetilde{c}, \widetilde{a}, \widetilde{\mathcal{B}}, \widetilde{t} \right\rangle \leftarrow$ MIPSolve($k_{MIP}, \widetilde{T}, \widetilde{l}, \widetilde{u}$)
6:      if $\widetilde{c}$ is Unsat then $c \leftarrow$ Replay($\widetilde{t}$)
7:      else if $\widetilde{c}$ is Sat then
8:          $a' \leftarrow$ ImportAssignment($\widetilde{a}$)
9:          $c \leftarrow$ ExactReseed($a', \widetilde{\mathcal{B}}$)
10:         if $c$ is Unknown then
11:             $c \leftarrow$ ExactSolve($+\infty$)
12:     if $c$ is Unsat or ($c$ is Sat and $a$ is integer-compatible) then
13:         return $c$
14:     else
15:         Generate a branching theory lemma using (4.3)
16:         return Unknown
```

Figure 5.4: The IntegerSolve procedure for linear integer arithmetic.

IntegerSolve which illustrates our approach. First, the real relaxation of the problem is solved using the BalancedSolve algorithm described above. If the real relaxation is unsatisfiable, then we are done. Otherwise, we construct an MIP instance and call an MIP solver (again with a heuristic pivot limit k_{MIP}) to search for an integer-compatible solution. When **Unsat** is returned, we also retrieve a *proof tree* $\widetilde{t}$, which is a record of the steps taken by the MIP solver to determine that the problem is integer-infeasible. The procedure attempts to verify the tree by *replaying* its proof in the exact solver using the Replay procedure described below. Otherwise, if **Sat** is returned, we attempt to verify the assignment as before. If the verification fails, we again call ExactSolve to ensure that we have a solution to the real relaxation before continuing. In the case that we are unable to verify that the problem is **Unsat** or do not find an integer-compatible assignment, we force a branch by generating a theory lemma of the form (4.3) and return: $x_i \leqslant \lfloor a_i \rfloor \vee x_i \geqslant \lceil a_i \rceil$.

5.2.1 MIP Proof Trees

We briefly described the major steps of a branch-and-cut MIP solver in Section 4.4.4. The reasoning of the MIP solver does internally to conclude that an input problem is infeasible (branches, cuts, and the current real-relaxation is infeasible) can be thought of as forming a *proof tree*. We now show how proof trees

$$\text{Propagate}\ \frac{\widetilde{\mathfrak{e}} \subseteq C^N \cup \widetilde{P} \quad \widetilde{\mathfrak{h}} \text{ is an inequality constraint} \quad \widetilde{\mathfrak{e}} \models_{\widetilde{T}} \widetilde{\mathfrak{h}}}{N_1 := N \cdot \left\langle \widetilde{\mathfrak{h}}, \widetilde{\mathfrak{e}} \right\rangle}$$

$$\text{Close}\ \frac{\widetilde{P} \cup C^N \models_{\widetilde{T}} \textbf{false}}{N_1 := N \cdot \bot}$$

$$\text{Branch}\ \frac{\widetilde{a} \text{ satisfies } \widetilde{P} \wedge C^N \quad x_i \in \mathcal{X}_{\mathbb{Z}} \quad \widetilde{a}_i = \alpha \quad \alpha \notin \mathbb{Z}}{N_1 := N \cdot \langle x_i \leqslant \lfloor \alpha \rfloor, \varnothing \rangle \quad \| \quad N_2 := N \cdot \langle x_i \geqslant \lceil \alpha \rceil, \varnothing \rangle}$$

Figure 5.5: Derivation rules. N is the parent node, N_1 and N_2 its child nodes. The symbol $\cdot$ denotes sequence concatenation.

extracted from the MIP solver can be replayed using the exact solver. For the rest of the section, let M be an MIP instance consisting of an LP problem P of the form $T\mathcal{X} = 0 \wedge l \leqslant \mathcal{X} \leqslant u$. Let $\widetilde{P}$ be the approximate version of P obtained by converting all the rational constants in P to their corresponding floating point constants.

The process that an branch-and-bound MIP solver goes through before concluding that $\widetilde{P}$ is integer-infeasible can be described at an abstract level as a search tree. The root node represents the initial problem $\widetilde{P}$ and each non-root node is derived from its parent by adding a constraint to the problem, either a cut or a branch. The leaves of the tree represent real-infeasible problems.

Formally, we define a tree node N as a sequence of pairs $\left\langle \widetilde{\mathfrak{h}}, \widetilde{\mathfrak{e}} \right\rangle$, where $\widetilde{\mathfrak{h}}$ is an inequality constraint (a half-plane) and $\widetilde{\mathfrak{e}}$ is an explanation, a (possibly empty) finite set, each element of which is either some $\widetilde{\mathfrak{h}}'$ where $\left\langle \widetilde{\mathfrak{h}}', \widetilde{\mathfrak{e}}' \right\rangle$ appears earlier in N or is a constraint from the initial problem $\widetilde{P}$. We denote by C^N the set of all constraints added along the path from the root node to N.

$$C^N = \left\{ \widetilde{\mathfrak{h}} \mid \left\langle \widetilde{\mathfrak{h}}, \widetilde{\mathfrak{e}} \right\rangle \in N \right\}$$

The root node of a proof tree is the empty sequence. Each non-root node is the result of applying to its parent node one of the derivation rules in Figure 5.5. The Propagate rule is used to record when the MIP solver adds a cut. The cut must be entailed by some subset of constraints in the current MIP problem. The cut and its explanation are recorded in the child sequence. We write $\widetilde{\mathfrak{e}} \models_{\widetilde{T}} \widetilde{\mathfrak{h}}$ to denote a currently untrusted claim that $\mathfrak{e} \models_{\mathbb{Z}\mathbb{R}} \mathfrak{h}$. The Branch rule is used to record when the MIP solver does a case split on an integer variable. This can happen when the MIP solver has a solution $\widetilde{a}$ to the real relaxation of the current problem that is not integer-compatible. The MIP solver may choose an integer variable x_i that has been assigned a real value α and enforce the constraint

$$x_i \leqslant \lfloor \alpha \rfloor \vee x_i \geqslant \lceil \alpha \rceil .$$

The rule has two children, each of which records in its sequence one of the two branch cases (with an empty explanation). A node N is a leaf when the MIP solver concludes that the problem $\widetilde{P} \cup C^N$ is (real)-infeasible.

Ideally, a proof tree would allow us to prove that the original problem P is integer-infeasible. However, because of the approximate representation used by the MIP solver, this is not always the case. As a consequence, our theory solver uses the proof tree just as a guide for its own internal attempt to prove that P is integer-infeasible. This process is captured at a high level by the REPLAY function.

5.2.2 Replaying MIP Proof Trees

The REPLAY function is shown in Figure 5.6. It takes an initially empty sequence η and a proof tree t, and traverses the tree with the goal of computing and returning a conflict, a subset of the constraints in the original LP problem P that are integer-infeasible. When the top call of REPLAY returns the empty set it means the replay has failed.[2] As REPLAY traverses the tree, it constructs a sequence η which is analogous to the sequences in the tree nodes, except that it contains only those constraints that the internal exact solver has successfully replayed and so may only be a subset of those in the tree node.[3]

If t is a leaf node, then $\widetilde{P} \cup C^N$ should be integer-infeasible. We check the exact analog, $P \cup C^{\eta}$. If unsuccessful, we fail, returning $\varnothing$; otherwise, we return a conflict. To compute the conflict, we make use of an auxiliary function, REGRESS, which is not shown. REGRESS takes a conflict K and a sequence η of constraint-explanation pairs and recursively replaces any constraint in K by its explanation. The net effect is to ensure a conflict which is a subset of the constraints in P.

If the root of t has a single child, this child must have been derived using the Propagate rule. The last element of the sequence in the child node represents the new cut and its explanation. We convert the cut and its explanation to their exact analogs and then verify that we can derive the cut $\mathfrak{h}$ from the exact constraints

[2]For the purposes of its use in Figure 5.4 which is at a higher level of abstraction, a return value of the empty set should be considered **Unknown**, and any other return value should be considered **Unsat**.

[3] Once $\mathfrak{e} \models_{\mathbb{ZR}} \mathfrak{h}$ is concluded, the pair $\langle \mathfrak{h}, \mathfrak{e} \rangle$ is added to the current sub-tree. This is implemented using the constraints datastructure described in Section 2.2.4. This provides an implementation for regressing the derived conflicts into clauses.

```
procedure REPLAY($\eta, t$)
    $C^{\eta} \leftarrow \{\mathfrak{h} \mid \langle \mathfrak{h}, \mathfrak{e} \rangle \in \eta\}$
    if $t$ is a is a leaf node $N$ then
        // $t$ is an application of Close
        Construct $T, l, u$ from $P_{\mathsf{Relax}} \cup C^{\eta}_{\mathsf{Relax}}$
        $c \leftarrow$ BALANCEDSOLVE()
        if $c$ is Unsat then
            // BALANCEDSOLVE found a conflict
            Let $\psi_{\mathsf{Relax}} \subseteq P_{\mathsf{Relax}} \cup C^{\eta}_{\mathsf{Relax}}$ be a conflict
            return REGRESS($\psi, H$)
        else
            return $\varnothing$
    if the root of $t$ has only one child $c$ then
        // $t$ is an application of Propagate (a cut)
        $t' \leftarrow$ subtree of $t$ rooted at $c$
        $\langle \mathfrak{h}, \mathfrak{e} \rangle \leftarrow$ IMPORTCONSTRAINT($last(c)$)
        if $\mathfrak{e} \subseteq C^{\eta} \cup P$ and $\mathfrak{e} \models_{\mathbb{ZR}} \mathfrak{h}$ then
            return REPLAY($\eta \cdot \langle \mathfrak{h}, \mathfrak{e} \rangle, t'$)
        else
            return REPLAY($\eta, t'$)
    if the root of $t$ has two children $c_1$ and $c_2$ then
        // $t$ is an application of Branch
        for $i = 1, 2$ do
            $t_i \leftarrow$ subtree of $t$ rooted at $c_i$
            $\langle \mathfrak{h}_i, \varnothing \rangle \leftarrow last(c_i)$
            $K_i \leftarrow$ REPLAY($\eta \cdot \langle \mathfrak{h}_i, \varnothing \rangle, t_i$)
        $K \leftarrow$ REPLAYCONFLICT($K_1, \mathfrak{h}_1, K_2, \mathfrak{h}_2$)
        return REGRESS($K, H$)
```

Figure 5.6: The REPLAY procedure.

```
procedure ReplayConflict($\mathsf{K}_1, \mathfrak{h}_1, \mathsf{K}_2, \mathfrak{h}_2$)
    if $\mathfrak{h}_1 \in \mathsf{K}_1$ and $\mathfrak{h}_2 \in \mathsf{K}_2$ then
        // the conflicts $\mathsf{K}_1$ and $\mathsf{K}_2$ use the branches literals $\mathfrak{h}_1$ and $\mathfrak{h}_2$
        return $\mathsf{K}_1 \cup \mathsf{K}_2 \setminus \{\mathfrak{h}_1, \mathfrak{h}_2\}$ (perform resolution with $\mathsf{K}_1$, $\mathsf{K}_2$ and $\mathfrak{h}_1 \vee \mathfrak{h}_2$)
    else if $\mathfrak{h}_1 \notin \mathsf{K}_1$ and $\mathfrak{h}_2 \in \mathsf{K}_2$ then
        return $\mathsf{K}_1$// either $\mathsf{K}_1$ is a conflict not containing $\mathfrak{h}_1$ or $\mathsf{K}_1 = \varnothing$
    else if $\mathfrak{h}_1 \in \mathsf{K}_1$ and $\mathfrak{h}_2 \notin \mathsf{K}_2$ then
        return $\mathsf{K}_2$
    else
        return $\varnothing$ // no conflict was found
```

Figure 5.7: The ReplayConflict procedure.

in $\mathfrak{e}$. These steps are explained in more detail in Section 5.2.3. If the cut can be verified, it and its explanation are included in the parameter η passed to the next recursive call to Replay. If not, the recursive call is made without $\mathfrak{h}$ in the hopes that it is not needed to derive a conflict.

The final case is when the root of t has two children, indicating that the Branch rule was applied. Because branch constraints only use integers, importing them cannot fail. We are always able to represent them exactly. Thus, we simply call Replay recursively on each of the two sub-trees, passing one of the branch conditions to each sub-tree.

We construct a conflict from the two conflicts returned by both of the subtrees (K_1 and K_2) using the ReplayConflict procedure (Figure 5.7). If both conflicts use their branch literals $\mathfrak{h}_1$ and $\mathfrak{h}_2$, then $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are removed by performing resolution (1.4) with K_1, K_2 and $\mathfrak{h}_1 \vee \mathfrak{h}_2$. If the sub-tree for K_1 succeeded without using $\mathfrak{h}_1$, then K_1 is a conflict for the current tree. (The K_2 case is similar.) Otherwise, the procedure fails to find a conflict for this tree and $\varnothing$ is returned. Note that the case where only one branch is needed enables backtracking while the conflict K_i does not use the current branch literal. This is essentially *non-chronological backtracking* [101].

Multiple Conflicts The implementations of Simplex given in Chapters 2 and 3 are capable of finding multiple conflicts in a single call. As we do not have any intuition into which conflicts are likely to lead to the proof of the infeasibility of root node, we try to make use of all of the conflicts. It is straightforward to extend Replay to return sets of conflicts per tree instead of a single conflict per tree. The procedure ReplayConflict then becomes two rounds of the Davis-Putnam procedure to remove $\mathfrak{h}_1$ and $\mathfrak{h}_2$ (Section 1.3). The Davis-Putnam procedure can potentially create an exponential number of clauses. To control the blowup in the number resulting clauses, CVC4 heuristically employs a full round of *subsumption* checking. A clause C is subsumed by a clause C' if $C \subset C'$. While subsumption checking is potentially expensive, this seems to pay for itself very quickly in practice. In the case that subsumption checking alone does not curtail the blowup, the solver is free to keep only some of the conflicts.[4]

5.2.3 Verifying Cuts During Replay

Lines 14 and 15 of REPLAY require converting $\left\langle \widetilde{\mathfrak{h}}, \widetilde{\mathfrak{e}} \right\rangle$ to an exact analog, $\langle \mathfrak{h}, \mathfrak{e} \rangle$, and then verifying that $\mathfrak{h}$ can be derived from $\mathfrak{e}$. We have implemented support for both Mixed-Gomory cuts and a variant of Mixed Integer Rounding cuts [79, 80, 88], but here we will only explain how reconstruction works for a special case of Gomory cutting planes (Section 4.4.3).

The MIP solver can add a Gomory cutting plane $\widetilde{\mathfrak{h}}$ when the following conditions hold:

(i) there is a row in $\widetilde{T}$ where x_i is basic ($x_i = \sum \widetilde{T_{i,j}} \cdot x_j$);

(ii) all of the non-basic variables on the row are assigned to either their upper or lower bound;

(iii) a subset of the variables on the row, that must include the basic variable x_i, are integer variables; and

(iv) the assignment of x_i is non-integer ($\widetilde{a}_i \notin \mathbb{Z}$).

The premises (i)-(iv) make up the explanation $\widetilde{\mathfrak{e}}$. (See [47] for the derivation of this cut and a Gomory cutting plane rule that does not use additional assumptions.) For simplicity of presentation, we additionally assume all of the variables are integer and all the coefficients $\widetilde{T_{i,j}}$ are positive ($\widetilde{T_{i,j}} > 0$) and assigned to their upper bounds, ($\widetilde{a}_j = \widetilde{u}_j$). The assignment to x_i is then determined by the upper bounds of the non-basic variables, $\widetilde{a}_i = \sum \widetilde{T_{i,j}} \widetilde{u}_j$. The cut $\widetilde{\mathfrak{h}}$ for these constraints is then[5]

$$\sum \frac{\widetilde{T_{i,j}}}{\widetilde{a}_i - \lfloor \widetilde{a}_i \rfloor} \left(\widetilde{u}_j - x_j \right) \geqslant 1.$$

Given $\left\langle \widetilde{\mathfrak{h}}, \widetilde{\mathfrak{e}} \right\rangle$ and the knowledge that the Gomory cutting plane procedure was used, we can attempt to derive a trusted cut and explanation $\langle \mathfrak{h}, \mathfrak{e} \rangle$ as follows. For the cut to be reconstructed, for every bound $x_j \leqslant \widetilde{u}(x_j) \in \widetilde{\mathfrak{e}}$ there must

[4] Subsumption checking was the much more successful of these two heuristics. Examples that previously ran out of memory began returning a maximum of 8 conflicts per sub-tree. The heuristic to discard conflicts was never triggered in the experiments in Section 5.3.

[5] This is using a stronger Gomory rule than the one proven in Section 4.4.3. See [47] for the derivation of this cut.

be a corresponding bound $x_j \leqslant u(x_j)$ in the exact system. (Note: $x_j \leqslant u(x_j)$ can be in either P or C^η.) Next we attempt to reconstruct the row

$$x_i = \sum \widetilde{T_{i,j}} x_j$$

in exact precision as a row vector z. The coefficient for the basic variable in z is -1 ($z_i = -1$). Nonbasic variables' coefficients are estimated from the approximate variables, $z_j = \textsc{DioApprox}(\widetilde{T_{i,j}}, D)$. If after approximation, the sign of z_j does not match the sign of $\widetilde{T_{i,j}}$, this cut cannot be reproduced. (Crucially, this includes the equals to 0 case.) The equalities $T\mathcal{X} = 0$ entail $\sum z_k x_k = 0$ iff z is in the row span of T. This entailment can be checked by replacing auxiliary variables with their original definitions, ($A_k \cdot \mathcal{X} = 0$), to get:

$$z_i \cdot x_i + \sum_{x_j \text{ is structural}} z_j x_j + \sum_{x_k \text{ is auxiliary}} \left(z_k x_k + z_k A_k \cdot \mathcal{X} \right).$$

The cut is rejected if any of the coefficients do not cancel to zero. The row vector z and the bounds u_j are used to generate $f_0 = \sum z_j u_j$, which can be thought of as a potential assignment to x_i. The cut cannot be reproduced if $b \in \mathbb{Z}$. If the value of f_0 is non-integer, the Gomory cut $\mathfrak{h}$:

$$\mathfrak{h} : \sum \frac{z_j}{f_0 - \lfloor f_0 \rfloor} (u_j - x_j) \geqslant 1$$

has been reproduced in exact precision. The explanation $\mathfrak{e}$ for the half plane $\mathfrak{h}$ includes the upper bounds $x_j \leqslant u_j$, and the equations $A_k \cdot \mathcal{X} = 0$ for the auxiliary variables (with $z_k \neq 0$).

Alternatively, z can be generated by Gaussian elimination starting from the equalities $A \cdot \mathcal{X} = 0$. Let Γ be the set of auxiliary variables appearing on $\widetilde{T_i}$.

$$\Gamma = \left\{ j \middle| \widetilde{T_{i,j}} \neq 0, j \in \text{Aux} \right\}$$

First, we generate a sub-matrix of A using the rows A_j for all j $in\Gamma$. Next we solve for the tableau form of these rows where x_i is basic and x_j is non-basic for all $j \neq i$ and $j \in \Gamma$. (This orientation of basic and non-basic variables may not be possible.) If successful, we will have regenerated the row $\widetilde{T_i}$ in exact precision as z. The implementation in CVC4 uses this Gaussian elimination method as a backup to the "guess and check" method of generating z.

5.3 Experiments

All of the algorithms in this paper have been implemented in the CVC4 SMT solver [7].[6] In this section, we report the results of experiments using these implementations.

5.3.1 Implementation Details and Heuristics

There are two different simplex implementations in CVC4, one that follows the well-known simplex adapted for SMT described in [46, 47], and one based on sum-of-infeasibilities as described in [73]. The experiments were run using the latter method for the EXACTSOLVE procedure with a pivot cap of $k_{EX} = 200$ in Fig. 5.1 (with $k_{FI} = 200$ for non-final calls). Values of other parameters used in our experiments are $D = 2^{26}$; $\epsilon = 10^{-9}$; $k_{LP} = 10000$; and $k_{MIP} = 200000$. For both the LP and MIP solvers, we use the floating-point simplex solver in GLPK version 4.52 [78], instrumented to communicate the additional information needed by CVC4 in order to verify assignments, conflicts, and proof trees.[7]

To avoid branching loops in GLPK, GLPK is halted if it branches 100 times on any one variable. To keep the size of the numeric constants manageable, we reject any cut containing a coefficient $\frac{n}{d}$ where $\log_2(|n|) + \log_2(|d|) > 512$. Further, we have a heuristic that dynamically disables the GLPK solver if it claims the problem is real-feasible and then integer-infeasible without generating any branches or cuts, a strange situation that happens with the `convert` benchmarks (see discussion below for details). GLPK is also dynamically disabled if CVC4's bignum package throws an exception while trying to import a floating point number. CVC4 has a heuristic that automatically detects and re-encodes benchmarks in the `QF_LRA` family miplib (which are derived from benchmarks in [2]) in something closer to their original form.[8]

5.3.2 Empirical Results

The experiments were conducted on the StarExec platform [104] with a CPU time limit of 1500 seconds and a memory limit of 8GB. We performed a comparison of our implementation with other SMT solvers over the full sets of `QF_LRA`

[6] Experiments were run using a branch of CVC4 available at `github.com/timothy-king/CVC4/CVC4` (commit 2550b6d).

[7] Source for this modified version of GLPK is available at `github.com/timothy-king/glpk-cut-log` (commit a35b8e).

[8]We did compare other solvers on the miplib problems after this re-encoding and the results were similar to those reported in Table 5.1: the re-encoding does not seem to help other solvers much.

and QF_LIA benchmarks from the SMT-LIB library (the "March 7 2013" version on StarExec), as well as the latendresse QF_LRA benchmarks from [73].

Table 5.1 gives the results for both QF_LRA and QF_LIA. The first segment of Table 5.1 summarizes the results over all of the problems in the categories. The QF_LIA benchmarks are additionally divided into the conjunctive subset and the non-conjunctive subset. The conjunctive subset consists of all families, all of whose benchmarks are a simple conjunction of constraints.[9]

The primary experimental comparison is between a configuration of CVC4 running just its internal solvers ("CVC") against a configuration with the techniques of this paper enabled ("CVC4+MIP"). Across all benchmarks, we also compare against similar state-of-the-art SMT solvers: mathsat5 (smtcomp12 version) [27], z3 (v4.3.1) [38], and yices2 (v2.2.0) [45]. We include a comparison against the version of AltErgo [16] used in [17] on just the QF_LIA benchmarks. For the conjunctive subset, we also give results for several solvers that support only conjunctive benchmarks: cutsat (CADE11) [69], SCIP (scip-3.0.0-ex+spx) [1, 33], and glpk (4.52) [78]. This version of SCIP handles MIP problems in exact precision.

To focus on the effects of the techniques proposed in this chapter, we also report only the results on benchmarks for which CVC4+MIP invokes GLPK at least once. In Table 5.1, the second segment contains the results for the QF_LRA benchmarks, the results for the non-conjunctive QF_LIA benchmarks are in the third segment. and the fourth segment contains the results for the conjunctive QF_LIA benchmarks. For these results, the second column of numbers indicates how many benchmarks in the family are included in the results. (See http://cs.nyu.edu/~taking/fmcad14_selections for a list of selected benchmarks.)

To better understand how successful the verification and replaying algorithms for integers described in Section 5.2 are, we analyzed all of the QF_LIA instances which were solved by CVC4+MIP and for which MIPSolve was invoked at least once, and collected the following statistics: the number of times MIPSolve was called, the number of attempts and successes at verifying **Sat** results from MIPSolve, and the number of attempts and successes at replaying **Unsat** results from MIPSolve. The results are shown in Table 5.2.

5.3.3 Discussion

On QF_LRA benchmarks, CVC4+MIP solves all of the problem instances that the already competitive CVC4 does plus 9 additional problems (solving more

[9]The conjunctive families are dillig, miplib2003, prime-cone, slacks, CAV_2009, cut_lemmas, pidgeons, and pb2010. For comparison purposes, we also translated them into the SMT-LIBv1.0 and MPS formats. The translations are available at http://cs.nyu.edu/~taking/conjunctive_integers.tbz.

than any other solver), all from the challenging miplib family. After preprocessing, these benchmarks are represented internally as mixed linear real and integer problems, so the INTEGERSOLVE procedure is used. CVC4+MIP solves the `opt1217--{27,37,57}.smt2` benchmarks in about 1s each and is the only solver to solve these benchmarks. These and a handful of other miplib problems are real-infeasible and are solved very quickly by BALANCEDSOLVE. INTEGERSOLVE is able to verify that several other miplib benchmarks are **Sat**. It was not able to successfully solve the most difficult problems which are real-feasible but integer-infeasible.

CVC4+MIP is also quite competitive on the `QF_LIA` problem instances. Particularly dramatic is the improvement of CVC4+MIP over CVC4 on the (related) families dillig, slacks, and CAV_2009_benchmarks. These benchmarks are small, randomly generated, conjunctive problems that are mostly satisfiable [44, 69]. It appears from Table 5.2 that CVC4+MIP does well on these families due to a high proportion of successes when IMPORTASSIGNMENT and EXACTRESEED are used to verify **Sat** instances. Excluding the convert family, GLPK returned **Sat** 1203 times, and in 1057 cases, we were able to verify this with the exact solver. Given the challenges of implementing branching and cutting within SMT solvers, this suggests that the technique of soundly verifying results from an external solver offers a new powerful tool in designing `QF_LIA` solvers. The empirical results on the REPLAY procedure, while not as dramatic, are also promising. Excluding the convert benchmarks, REPLAY was successful on 425 out of 652 invocations and did particularly well on (relatively) easy benchmarks e.g. calypto and prime-cone.

CVC4+MIP is competitive with the dedicated conjunctive solvers we included. Of course, its performance is limited by that of GLPK (Interestingly, CVC4+MIP outperforms GLPK on these benchmarks.) This seems to be because the tableau presented by CVC4 to GLPK (particularly for the slacks benchmarks) is often better for GLPK than the initial one. Though most of the improvement of CVC4+MIP over CVC4 is on conjunctive benchmarks, this seems to be an artifact of the benchmarks.

The convert family is interesting in that almost every proof reported by GLPK on these benchmarks fails to replay. These benchmarks encode fixed-width bitvector problems using `QF_LIA`. The encoding of the bitvector select operator creates integer equalities between variables with coefficients of massively different scales. Consider the following example:

```
convert-jpg2gif-query-1471.smt:
  (= z231_31_0_ (+ z231_1_0_ (* 4 z231_31_2_)))
```

This encodes that a the variable $z231\_31\_0\_$ which represents a 32-bit bitvector is equal to its upper 30 bits, represented by $z231\_31\_2\_$, concatenated onto its lower

2 bits, represented by z231_1_0_.

The reason these benchmarks fail has to do with an internal heuristic in GLPK. To ensure numerical stability, GLPK increases each bound by some amount ϵ, where ϵ is proportional to the size of the bound. Because of the dramatic differences of scale in the coefficients in the convert family, GLPK increases some bounds by a large amount and others by a small amount. As a result, GLPK frequently makes incorrect conclusions (both feasible and infeasible) about subproblems from this family. These benchmarks thus present a challenge for the techniques given in section 5.2 and are a good subject for future research.

5.4 Future Work

The results of this chapter suggest several lines of future research for integrating Simplex floating point simplex solvers in the context of SMT. The work of this chapter establishes a beneficial integration of an MIP solver into an SMT solver. The work has been strongly biased towards not interfering with the search of the MIP solver and only adding new logging features to the MIP solver.

Optimization Modulo Theories This work has integrated both a LP solver and a MIP solver within an SMT solver for feasibility checking. A potential advantage of this is that the SMT solver can take advantage of the other features of the LP and MIP solver. The theory solver can now use these as subroutines for performing *Optimization Modulo Theories* [100]. The rough outline of an Optimization Modulo Theories solver is very similar to the branch-and-bound strategy in Section 4.4.4. The solver is given an additional user command (`minimize t`) where t is a term in an totally ordered domain. (For simplicity, assume $>$ is the order.) The solver first finds some satisfying interpretation M that evaluates t to the value t^M. An outer loop records the value of t^M and adds the assertion

$$\left(\textbf{assert}\ \left(<\ t\ t^M\right)\right)$$

to force a new model with a better assignment to be found. An immediate improvement is to first find an initial model for the current SAT solver assignment and to next call either the LP solver or MIP solver to optimize t [77,100].[10]

Safely Generated Cuts and Conflicts The papers [32,89] describe generating exact precision cuts and conflicts starting from floating point representations.

[10] The simple loop is a bit too naive to terminate if t : `Real` or t : `Int`.

This is done by using an exact representation of the $A\,\mathcal{X} = 0$ constraints mixed with directional rounding of floating point numbers. Showing that the real-relaxation is infeasible is naturally expressible as an instance of Farkas' lemma (Section 2.2.2). Roughly speaking, one can compute the instance of Farkas' lemma using directional rounding in each floating point computation to yield some $\widetilde{\gamma}$ that is entailed to be $0 > \widetilde{\gamma}$, but is actually $\widetilde{\gamma} > 0$. This allows a conflict to be extracted. If done soundly, the derived contradiction can then be used by the SMT solver. Implementing these techniques appears to require a new implementation for every cutting plane technique and application of Farkas' lemma in the implementation. These results could then be used during the REPLAY procedure if they are stored on leaves of the MIP proof trees.

Arbitrary Precision Floating-Point LP Solvers The state-of-the-art exact precision LP and MIP solvers additionally implement LP solvers on top of fixed precision floating point numbers [3,33]. (Fixed precision means roughly that the floating point numbers can have k bits of representation for any $k > N$.) When the fixed precision floating point solver gives a result, this result is checked in an exact precision solver (much in the same way EXACTRESEED does). If the fixed precision solver does not give a correct result, the precision of the solver is increased, and the floating point solver is invoked again. This process is repeated until an exact precision result is obtained. This may be a way around the tolerance problems discovered by the convert benchmarks.

Exact Precision LP Solvers One line of research would be to directly use an out-of-the-box exact precision LP or MIP solver to avoid implementing the advanced techniques they contain. The work in [33] describes a tool called Exact MIP which is implemented in SCIP. However, due to its license, this program is unappealing for integration with CVC4. The tool QsOpt_ex described in [3] may be appropriate for future experiments. The GLPK framework our tool currently supports does have a an exact precision LP solver, GLP_EXACT. However, this currently takes its input in floating point form. While this can be overcome, it needs to be noted that GLPK's exact precision LP solver cannot be used as the underlying LP solver to GLPK's MIP solver.

Tight Integration of REPLAY This integration of the theory solver and the MIP solver in REPLAY is intentionally quite loose. The interaction with the MIP solver is through callbacks and requests for information to log the proof trees. An alternative to this is to implement REPLAY using a tight integration of the MIP solver and abstract steps that REPLAY is taking. These abstract steps may be thought of as an abstract calculus, much in the same vein as Abstract

DPLL [90] or the Model Constructing Calculus [40]. The decisions of this calculus are the branches, the Propagate rule adds entailed facts, and resolution is used for backtracking.[11] Instead of allowing the MIP procedure to proceed as normal, it would be forced to call back into the theory solver to check the correctness of real-infeasibility claims and cuts. This would allow for a large number of techniques from SMT solving to be implemented on top of the MIP solver. When claimed cuts fail to be validated, the theory solver can force the cut to be dropped.[12] When real-infeasibility claims fall through, the SMT solver knows that it is in an **Unknown** state and can start backtracking the solver to a state where a proof on **Unsat** may still be derived. Further, the theory solver could cache lemmas between branches and use these to implement BCP to strengthen the MIP solvers state and reduce redundant search. The theory solver can also implement non-chronological backtracking so that sub-trees unnecessary to proving infeasibility are not explored. Note that such a level of control has replaced all aspects of the MIP solver's branch-and-cut loop. The only major exception is the selection of branches. A side benefit is that tight integration like this could potentially cut down memory usage by only storing a single path of the MIP tree instead of the full tree. (This is somewhat at odds with the lemma caching+BCP suggestion.) Memory usage has not yet been a problem in practice.

Strict Inequality Encoding One of the many lessons of the failure on the convert family is that the current encoding for strict inequalities is likely to be too naive. The encoding transforms $x \leqslant_\delta d + e\delta$ into $x \leqslant d + e\epsilon$ where ϵ is a small fixed constant. In the implementation, ϵ is 10^{-9}. (Note that this affects only bounds on real variables as bounds on integer variables are always rounded to integer values (Section 4.3).) This is an appealing heuristic as it is both simple and it matches the selection of a small enough value of β in Section 2.2.11. As part of these experiments, we discovered the role of the tolerances in the intermediate calculations. Changing the bound by a small, fixed amount ϵ is likely to be within tolerance for most variables and unlikely to change the choices of whether or not a variable is strictly above or at its bound. One possible way to fix this problem is to scale the bounds of each variable independently to take tolerance for that bound into account. Another possibility is to encode δ as an explicit variable. The LP solver would be run with the constraint that $\delta \geqslant 0$. If the LP solver found that the real-relaxation was feasible, but the assignment to δ was not > 0, the LP solver would be asked to optimize δ. If the LP solver finds

[11] What would distinguish this calculus is this addition of all of the operations potentially being unsuccessful.

[12] Alternatively, the theory solver is the one generating the cuts so that they are correct by construction. The theory solver would then give approximate versions to the MIP solver.

the optimal feasible answer[13] as $\delta = 0$, the LP solver should be in a state such that the row for the optimization function $\delta = \sum_{j \in \mathcal{N}} c_i x_j$ can prove that $\delta \leqslant 0$ via an instantiation of Farkas' lemma. By recreating this state in the theory solver, we can attempt to prove the conflict.

[13] Many LP solvers can be parameterized to stop once the optimization function is over a constant threshold. In this case, the solver can stop once $\delta > C$ for some small C.

			CVC4+MIP		CVC4		yices2		mathsat5		Z3		altergo		cutsat		scip		glpk	
set	# inst.	# sel.	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
Selecting all benchmarks in the family																				
QF_LRA	652	652	645	6966	636	8557	632	5350	622	10913	615	5696	-	-	-	-	-	-	-	-
¬C. QF_LIA	4579	4579	4489	86854	4472	86375	4375	30656	4543	55417	4474	75171	3956	262031	-	-	-	-	-	-
C. QF_LIA	1303	1303	1249	11130	1068	31054	1111	55691	1154	33260	1039	19015	1232	2055	1018	35330	1255	7164	1173	8895
total	6534	6534	6383	104950	6176	125986	6118	91697	6319	99590	6128	99882	-	-	-	-	-	-	-	-
Selecting QF_LRA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
miplib	42	37	30	1530	21	3037	23	2730	17	5682	18	2435	-	-	-	-	-	-	-	-
DTP-…	91	4	4	4	4	4	4	0	4	2	4	1	-	-	-	-	-	-	-	-
latendresse	18	18	18	767	18	836	12	85	10	99	0	0	-	-	-	-	-	-	-	-
total	-	59	52	2301	43	3877	39	2815	31	5783	22	2436	-	-	-	-	-	-	-	-
Selecting non-conjunctive QF_LIA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
convert	319	282	208	9646	193	9343	188	4337	274	1876	282	118	166	272	-	-	-	-	-	-
bofill-…	652	460	460	5401	458	4490	460	748	460	1519	460	2060	67	55	-	-	-	-	-	-
CIRC	51	11	11	0	11	0	11	0	11	0	11	0	11	0	-	-	-	-	-	-
calypto	37	37	37	3	37	3	37	0	37	6	36	5	35	24	-	-	-	-	-	-
nec-smt	2780	207	207	17276	207	18045	199	777	207	17925	201	7209	184	23724	-	-	-	-	-	-
wisa	5	1	1	0	1	0	1	0	1	1	1	0	1	0	-	-	-	-	-	-
total	-	998	924	32326	907	31881	896	5862	990	21327	991	9392	464	24075	-	-	-	-	-	-
Selecting conjunctive QF_LIA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
dillig	233	189	189	49	157	9823	175	8557	188	7185	166	1269	189	5	166	5840	189	42	189	3
miplib2003	16	8	4	307	4	1283	4	507	5	354	5	1089	0	0	6	146	7	17	6	295
prime-cone	37	37	37	2	37	2	37	2	37	1	37	2	37	1	37	4	37	1	37	0
slacks	233	188	166	61	93	2003	107	15672	119	4741	90	1994	188	84	96	6324	161	2361	101	11
CAV_2009	591	424	424	69	346	10035	376	26351	421	10236	354	2759	423	323	377	17015	424	105	424	6
cut_lemmas	93	74	62	9581	64	6865	72	1662	45	9472	38	5858	74	267	15	1887	72	1757	71	760
total	-	920	882	10069	701	30011	771	52751	815	31989	690	12971	911	680	697	31216	890	4283	828	1075

Table 5.1: Experimental results on `QF_LRA` and `QF_LIA` benchmarks. C. `QF_LIA` stands for Conjunctive `QF_LIA` and ¬C. `QF_LIA` standing for Not Conjunctive `QF_LIA`.

			Sat		Unsat	
set	# sel.	MIPSOLVE calls	attempts	successes	attempts	successes
QF_LIA	1393	3873	2559	1058	652	425
convert	208	2130	1356	1	178	3
bofill-scheduling	254	254	245	245	0	0
CIRC	11	85	6	5	79	77
calypto	37	375	77	23	293	278
wisa	1	1	1	1	0	0
dillig	189	228	225	185	3	2
miplib2003	4	10	3	3	5	4
prime-cone	37	37	19	19	18	18
slacks	166	195	168	162	3	3
CAV_2009	424	469	459	414	8	7
cut_lemmas	62	89	0	0	65	33

Table 5.2: Success rate of reproducing results of MIPSOLVE.

Chapter 6

Conclusions

The design and implementation of theory solvers critically affect the overall efficiency of SMT tools. As SMT solvers grow in prominence in automated formal methods so too does the importance of designing efficient theory solvers for theories of interest.

This thesis has presented techniques to improve state-of-the-art implementations of theory solvers for three of the core logics of SMT: quantifier-free linear integer and real arithmetics and their combination. This work builds upon the Simplex for SMT algorithm for `QF_LRA`, as well as known techniques to extend the `QF_LRA` theory solver to support the `QF_LIA` and `QF_LIRA` logics. This thesis gives an in-depth description of a theory solver implementing the Simplex for SMT algorithm. We propose two significant improvements to the core algorithm through better conflict detection and conflict strengthening and describe well known [but not well documented] techniques for implementing theory propagation and handling the internal arithmetic used by the algorithm. We also give experimental evidence showing that one of the key strengths of this decision procedure is keeping the number of pivots low for most check calls.

A weakness of the Simplex for SMT architecture is that its localized reasoning, while efficient for a large class of problems, can fail to quickly converge to a solution on classes of challenging benchmarks. To improve upon the robustness of the SMT solver, we have proposed an alternative Simplex based decision procedure that minimizes the sum of infeasibilities function. We give evidence that this new algorithm is competitive with the Simplex for SMT and is more robust on challenging problem instances.

To accelerate the exact precision theory solver, we present techniques for leveraging the results of linear programming and mixed integer programming solvers. Additionally, we show strongly positive results for using such solvers for certain families of challenging problems without compromising correctness of the SMT solver. Previous efforts to leverage such solvers in the context of

SMT concluded that such solvers are inappropriate for the context of SMT.

The experimental results show that the combination of the novel techniques presented here enable CVC4 to solve the most available `QF_LRA` and `QF_LIA` SMT-LIB benchmarks of any SMT solver for these logics in a reasonable amount of time.

To conclude, the contributions in this thesis have improved the state of the art in developing linear arithmetic solvers for SMT. These are three of the core logics in SMT solving and have some of the most mature implementations. The development of improved implementations and new algorithms for these logics has practical importance to automated formal methods. Hopefully, this research can point the way towards future breakthroughs in decision procedures for these theories.

Appendix A

Discussion and Proofs of Miscellaneous Theorems

A.1 Discussion

A.1.1 Variables vs. Uninterpreted Constants

This thesis does not distinguish between the phrases *variable* and *uninterpreted constant*. New users of SMT solvers often find it confusing that all "variables" (in a formal sense) are bound either in quantifiers or macros. All "variables" (in an informal sense) are declared as uninterpreted constant symbols. A "variable" (in an informal sense) `x` of sort `Real` is added as a new 0-ary function symbol `x` of sort `Real` to Σ. This approach cleans up formally defining uninterpreted functions and model construction of theory combinations. However, this has the disadvantage of formally adding expansions of models of the theory into the discussion. For uninterpreted constants, every satisfying expansion is isomorphic to a satisfying interpretations. For the purposes of this thesis, this layer of indirection imparts little wisdom.

A.1.2 Splitting Disequalities

Instead of the lemma $t = d \iff (t \leqslant d \wedge t \geqslant d)$, CVC4 discharges $t \leqslant d \vee t \geqslant d$ for splitting disequalities. The original lemma results in three clauses,

$$t = d \implies t \leqslant d, \qquad t = d \implies t \geqslant d, \qquad (t \leqslant d \wedge t \geqslant d) \implies t = d$$

The third clause is what is not yet satisfied in the trichotomy axiom,

$$\neg(t \leqslant d) \vee \neg(t \geqslant d) \vee t = d.$$

The counter intuitive choice of preferring $t \leqslant d \vee t \geqslant d$ to trichotomy is that this allows for two relaxations in the SAT solver and the combination framework: the CVC4 arithmetic theory solver does not have to be guaranteed that it is informed about all equalities over the type `Real`, and the SAT solver does not need to assign all SAT literals. Discharging the tricotomy lemma would lead to a soundness error. This is because $t = d$ may already be asserted to another theory (say uninterpreted functions) while theory combination generated an assertion $t + x \neq x + d$ that while semantically equivalent is not syntactically identical to $\neg(t = d)$, and hence it can be sent to arithmetic and $t = d$ can be asserted to the theory engine at the same time. The SAT solver can report all clauses as being satisfied without re-invoking arithmetic. On the other hand, in the current combination framework the arithmetic must be notified of a choice that satisfies $t \leqslant d \vee t \geqslant d$. Properly supporting disequalities in such a framework then also requires the solver to internally perform the following inference internally for soundness:

$$\frac{t \leqslant d \qquad t \neq d}{t < d}\ \textsc{Strict} \tag{A.1}$$

We leave it to the reader to perform the necessary changes to AssertUpper, AssertLower, and AssertDisequality to support this efficiently.

A.2 Proofs of Miscellaneous Theorems

A.2.1 From Delta Satisfaction to Densely Satisfied Regions

Revisiting Lemma (2.5). *If an extended assignment $a_\delta \models_{\delta\mathbb{R}} q$, then there exists an $\alpha > 0$ such that for all $\beta \in (0, \alpha)$ the following holds $M_{a_\delta,\beta} \models_{\mathbb{R}} p$.*

Proof. Suppose $a_\delta \models_{\delta\mathbb{R}} q$. The form of q is $t \bowtie_\delta d_q + e_q\delta$. Suppose that the value of the left-hand side t evaluates to $d_t + e_t\delta$ under a_δ in delta-arithmetic, so that $d_t + e_t\delta \bowtie_\delta d_q + e_q\delta$. As the trichotomy property holds for $<_\delta$. the constants $d_t + e_t\delta$ and $d_q + e_q\delta$ are related by either $<_\delta$, $=_\delta$ or $>_\delta$. We show in these three cases that there $\exists\alpha > 0$ s.t. $\forall\beta \in (0, \alpha)$ that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.

- Suppose $d_t + e_t\delta >_\delta d_q + e_q\delta$ holds. Either $d_q > d_t$ or $d_t = d_q$ and $e_t > e_q$ holds. The symbol $\bowtie_\delta$ is either $\geqslant_\delta$ or $\neq_\delta$, and the relational symbol in p ($\bowtie$) is either $\geqslant$, $>$, or $\neq$. As $\bowtie$ is not $<$, $e_q \geqslant 0$.
  - Suppose $e_t < e_q$. Both $e_q - e_t > 0$ and $d_t - d_q > 0$ hold. Let $\alpha =$

$\frac{d_t - d_q}{e_q - e_t} > 0$. Then, for all $\beta \in (0, \alpha)$,

$$\beta < \frac{d_t - d_q}{e_q - e_t} \quad \Longrightarrow \quad (e_q - e_t)\beta < d_t - d_q.$$

So, $d_t + e_t\beta > d_q + e_q\beta$, and $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.

- Suppose $e_t \geqslant e_q$. Then under the assumption that $d_t + e_t\delta >_\delta d_q + e_q\delta$ holds, $d_q > d_t$ must hold. Let α be any positive real number. Then, for all $\beta \in (0, \alpha)$,

$$(e_q - e_t)\beta \leqslant 0 < d_q - d_t.$$

So, $d_t + e_t\beta > d_q + e_q\beta$, and $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.

In both cases, $\exists\alpha > 0.\forall\beta \in (0, \alpha)$ such that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$.

- Suppose $d_t + e_t\delta <_\delta d_q + e_q\delta$ holds. (This case is symmetric analogous to the previous one.) Either $d_q < d_t$ or $d_t = d_q$ and $e_t < e_q$ holds. The symbol $\bowtie_\delta$ is either $\leqslant_\delta$ or $\neq_\delta$, and the relational symbol in p ($\bowtie$) is either $\leqslant$, $<$, or $\neq$. As $\bowtie$ is not $>$, $e_q \leqslant 0$.

  - Suppose $e_t > e_q$. Both $e_q - e_t < 0$ and $d_t - d_q < 0$ hold. Let $\alpha = \frac{d_t - d_q}{e_q - e_t} > 0$. Then, for all $\beta \in (0, \alpha)$,

$$\beta < \frac{d_t - d_q}{e_q - e_t} \quad \Longrightarrow \quad (e_q - e_t)\beta > d_t - d_q.$$

  So, $d_t + e_t\beta < d_q + e_q\beta$, and $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.

  - Suppose $e_t \leqslant e_q$. Then under the assumption that $d_t + e_t\delta <_\delta d_q + e_q\delta$ holds, $d_q < d_t$ must hold. Let α be any positive real number. Then, for all $\beta \in (0, \alpha)$,

$$(e_q - e_t)\beta \geqslant 0 > d_q - d_t.$$

  So, $d_t + e_t\beta < d_q + e_q\beta$, and $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.

  In both cases, $\exists\alpha > 0.\forall\beta \in (0, \alpha)$ such that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$.

- Suppose $d_t + e_t\delta =_\delta d_q + e_q\delta$ holds. Both $d_q = d_t$ and $e_t = e_q$ hold. The symbol $\bowtie_\delta$ is either $\geqslant_\delta$, $\leqslant_\delta$ or $=_\delta$, and the relational symbol in p ($\bowtie$) is either $\leqslant$, $<$, $>$, $\geqslant$ or $=$. Let α be any positive real number. We now show that for all $\beta \in (0, \alpha)$ that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ holds.

– Suppose $e_q = 0$. Then $\bowtie$ is either $\leqslant$, $\geqslant$ or $=$. Therefore

$$d_t + e_t\beta = d_q + e_q\beta = d_q,$$

and $M_{a_\delta,\beta} \models_{\mathbb{R}} t \bowtie d$ holds.

– Suppose $e_q > 0$. Then $\bowtie$ is $>$. Therefore

$$d_t + e_t\beta = d_q + e_q\beta > d_q,$$

and $M_{a_\delta,\beta} \models_{\mathbb{R}} t > d$ holds.

– Suppose $e_q < 0$. Then $\bowtie$ is $<$. Therefore

$$d_t + e_t\beta = d_q + e_q\beta < d_q,$$

and $M_{a_\delta,\beta} \models_{\mathbb{R}} t < d$ holds.

In all cases, $M_{a_\delta,\beta} \models_{\mathbb{R}} p$. So there $\exists\alpha > 0.\forall\beta \in (0,\alpha)$ such that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$.

□

A.2.2 From Densely Satisfied Regions to Delta Satisfaction

This section contains formalization and proofs for the reverse directions of Lemmas 2.5 and 2.6. For an extended assignment a_δ and a real value ζ, we denote by $a_{\zeta\delta}$ the assignment that scales all of the delta-coefficients in a_δ by ζ. If $a_\delta(x) = \langle d, e\rangle$, then $a_{\zeta\delta}(x) = \langle d, \zeta e\rangle$. We say that t is an *offset free* linear term if contains no sub-term $(t' + t'')$ such that t' or t'' is a rational constant symbol.

Claim. Let t be an offset free linear term. If the evaluation of t under a_δ is $\langle d, e\rangle$, then the evaluation of t under $a_{\zeta\delta}$ is $\langle d, \zeta e\rangle$.

Lemma A.1. *Let* p *have the form* $t \bowtie d$ *where* t *is an offset-free linear term and* q *be any admissible encoding of* p*. If there exists an* $\alpha > 0$ *such that for all* $\beta \in (0,\alpha)$ *the following holds* $M_{a_\delta,\beta} \models_{\mathbb{R}} p$*, then there exists a real value* $\eta \geqslant 1$ *such that for all* $\zeta \geqslant \eta$ *that* $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$.

Proof. Suppose there exists a delta-extended assignment a_δ and there exists $\alpha > 0$ such that for all $\beta \in (0,\alpha)$, $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ where p has the form $t \bowtie d$. The encoding q is any admissible conversion of p with the form $t \bowtie_\delta d_q + e_q\delta$. Let $d_t + e_t\delta$ be the evaluation of t under a_δ.

Given e_t and e_q, we define η in two cases.

$$\eta = \begin{cases} 1 & e_t = 0 \vee \frac{e_q}{e_t} \leqslant 1 \\ \frac{e_q}{e_t} & e_t \neq 0 \wedge \frac{e_q}{e_t} > 1 \end{cases}$$

Thus $\eta \geqslant 1$.

Let ζ be any real value greater than or equal to η. Under $a_{\zeta\delta}$, the evaluation of t is $\langle d_t, \zeta e_t\rangle$. Thus for each $\beta \in (0, \alpha)$, the evaluation of t is $d_t + e_t\zeta\beta$.

Our goal is show that $d_t + e_t\zeta\delta \bowtie_\delta d_q + e_q\delta$ must hold in all cases.

- Suppose $\bowtie$ is $>$ or $\geqslant$. Then q has the form $t \geqslant_\delta d_q + e_q\delta$ and $e_q \geqslant 0$.

  We know p is satisfied for some $\beta \in (0, \alpha)$ such that $d_t + e_t\beta \geqslant d_q$. By transitivity, $d_t + e_t\zeta\beta \geqslant d_q$.

  We first show that $d_t \geqslant d_q$.

  - Suppose for contradiction that $d_q > d_t$. Thus,

    $$e_t\zeta\beta \geqslant d_q - d_t > 0$$

    As $e_t\zeta\beta > 0$ and $\beta > 0$, it must be the case that $e_t\zeta > 0$. Thus $\beta \geqslant \frac{d_q - d_t}{e_t\zeta} > 0$.

    Let β' be any real value in the non-empty range $\left(0, \min\left(\alpha, \frac{d_q-d_t}{e_t\zeta}\right)\right)$. Thus $\beta' < \frac{d_q-d_t}{e_t\zeta}$ and $\beta' \in (0, \alpha)$. By the initial assumption, $M_{\alpha,\beta'} \models_{\mathbb{R}}$ p is satisfied. Thus $d_t + e_t\beta' \geqslant d_q$. By transitivity, $d_t + e_t\zeta\beta' \geqslant d_q$. Rewriting this we get that $\beta' \geqslant \frac{d_q-d_t}{e_t\zeta}$. This is a contradiction.

  Thus $d_t \geqslant d_q$.

  We now show that $\langle d_t, \zeta e_t\rangle \geqslant_{lex} \langle d_q, e_q\rangle$ holds in all cases.

  - Suppose $d_t > d_q$ holds. Then $\langle d_t, \zeta e_t\rangle >_{lex} \langle d_q, e_q\rangle$ holds.
  - Suppose $d_t = d_q$ and $\bowtie$ is $\geqslant$. Then $d_t + e_t\beta \geqslant d_q$ holds. Thus $e_t\zeta \geqslant 0$. As $0 = d_q$, $\langle d_t, \zeta e_t\rangle >_{lex} \langle d_q, e_q\rangle$ holds.
  - Suppose $d_t = d_q$ and $\bowtie$ is $>$. Then $d_t + e_t\beta > d_q$ holds as well as $d_t + e_t\zeta\beta > d_q$. Thus $e_t\zeta\beta > 0$ and $e_t > 0$ ($\zeta \geqslant 1$ and $\beta > 0$). Either $\eta = 1$ or $\eta > 1$.
    * If $\eta = 1$, then either $e_t = 0$ or $\frac{e_q}{e_t} \leqslant 1$. As $e_t > 0$, then $\frac{e_q}{e_t} \leqslant 1$. Thus $e_q \leqslant e_t$. This can be rewritten as $e_t\eta \geqslant e_q$. By transitivity, $e_t\zeta \geqslant e_q$. Thus $\langle d_t, \zeta e_t\rangle \geqslant_{lex} \langle d_q, e_q\rangle$ holds.
    * If $\eta > 1$, then $e_t \neq 0$ and $\frac{e_q}{e_t} > 1$. As $e_q > 0$, $e_t > 0$. Thus $\eta e_t = e_q$. By transitivity, $\zeta e_t = e_q$. $\langle d_t, \zeta e_t\rangle \geqslant_{lex} \langle d_q, e_q\rangle$ holds.

  $\langle d_t, \zeta e_t\rangle \geqslant_{lex} \langle d_q, e_q\rangle$ holds. In all of the above cases,

  $$d_t + \zeta e_t\delta \geqslant_\delta d_q + e_q\delta$$

  holds, and $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$.

- Suppose $\bowtie$ is $<$ or $\leqslant$. Then q has the form $t \geqslant_\delta d_q + e_q\delta$ and $e_q \leqslant 0$.

  We know p is satisfied for some $\beta \in (0, \alpha)$ such that $d_t + e_t\beta \leqslant d_q$. By transitivity, $d_t + e_t\zeta\beta \leqslant d_q$ ($\zeta \geqslant 1$).

  We first show that $d_t \leqslant d_q$.

  – Suppose for contradiction that $d_q < d_t$. Thus,

  $$e_t\zeta\beta \geqslant d_q - d_t > 0$$

  As $e_t\zeta\beta < 0$ and $\beta > 0$, it must be the case that $e_t\zeta < 0$. Because $d_q - d_t < 0$, $\beta \geqslant \frac{d_q - d_t}{e_t\zeta} > 0$ must hold.

  Let β' be any real value in the non-empty range $\left(0, \min\left(\alpha, \frac{d_q-d_t}{e_t\zeta}\right)\right)$. Thus $\beta' < \frac{d_q-d_t}{e_t\zeta}$ and $\beta' \in (0, \alpha)$. By the initial assumption, $M_{a,\beta'} \models_{\mathbb{R}}$ p is satisfied. Thus $d_t + e_t\beta' \leqslant d_q$. By transitivity, $d_t + e_t\zeta\beta' \leqslant d_q$. Rewriting this we get that $\beta' \geqslant \frac{d_q-d_t}{e_t\zeta}$. This is a contradiction.

  We now show that $\langle d_t, \zeta e_t\rangle \leqslant_{lex} \langle d_q, e_q\rangle$ holds in all cases.

  – Suppose $d_t < d_q$ holds. Then $\langle d_t, \zeta e_t\rangle <_{lex} \langle d_q, e_q\rangle$ holds.

  – Suppose $d_t = d_q$ and $\bowtie$ is $\leqslant$. Then $d_t + e_t\beta \leqslant d_q$ holds. Thus $e_t\zeta \leqslant 0$. As $0 = d_q$, $\langle d_t, \zeta e_t\rangle <_{lex} \langle d_q, e_q\rangle$ holds.

  – Suppose $d_t = d_q$ and $\bowtie$ is $>$. Then $d_t + e_t\beta < d_q$ holds as well as $d_t + e_t\zeta\beta < d_q$. Thus $e_t\zeta\beta < 0$ and $e_t < 0$ ($\zeta \geqslant 1$ and $\beta > 0$). Either $\eta = 1$ or $\eta > 1$.

    * If $\eta = 1$, then either $e_t = 0$ or $\frac{e_q}{e_t} \leqslant 1$. As $e_t < 0$, then $\frac{e_q}{e_t} \leqslant 1$ and $e_q \geqslant e_t$. This can be rewritten as $e_t\eta \leqslant e_q$. By transitivity, $e_t\zeta \leqslant e_q$. Thus $\langle d_t, \zeta e_t\rangle \leqslant_{lex} \langle d_q, e_q\rangle$ holds.
    * If $\eta > 1$, then $e_t \neq 0$ and $\frac{e_q}{e_t} > 1$. Thus $\eta e_t = e_q$. By transitivity, $\zeta e_t \leqslant e_q$. $\langle d_t, \zeta e_t\rangle \leqslant_{lex} \langle d_q, e_q\rangle$ holds.

  $\langle d_t, \zeta e_t\rangle \leqslant_{lex} \langle d_q, e_q\rangle$ holds. In all of the above cases,

  $$d_t + \zeta e_t\delta \leqslant_\delta d_q + e_q\delta$$

  holds, and $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$.

- Suppose $\bowtie$ is $=$. Then q has the form $t =_\delta d_q + e_q\delta$ and $e_q = 0$. We know p is satisfied for two distinct $\beta, \beta' \in (0, \alpha)$ such that

  $$d_t + e_t\beta = d_q \wedge d_t + e_t\beta' = d_q.$$

From this we can deduce that e_t must be 0, and $d_t = d_q$. Then for any value of ζ, the evaluation of t is invariant ($\langle d_q, 0\rangle$). Thus $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$ for any ζ.

- Suppose $\bowtie$ is $\neq$. Then q has the form $t \neq_\delta d_q + e_q\delta$ and $e_q = 0$. Either $d_t = d_q$ or not.
  - Suppose $d_t = d_q$. Then for some $\beta \in (0, \alpha)$, $d_t + e_t\beta \neq d_q$. Thus as $e_t\beta \neq 0$, $e_t \neq 0$. Then for any non-zero ζ, $\zeta e_t \neq 0$ and
  $$\langle d_t, \zeta e_t\rangle \neq \langle d_q, 0\rangle$$
  - Suppose $d_t \neq d_q$. Then for any ζ, $\zeta e_t \neq 0$ it is the case that
  $$\langle d_t, \zeta e_t\rangle \neq \langle d_q, 0\rangle$$

  Therefore $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$ in both cases for $\zeta \neq 0$.

Thus $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$ for some $\zeta \geqslant 1$. □

Next we generalize from literals to conjunctions of literals. Let the sequence of literals $p_1, \ldots, p_n$ generate the delta extended relations $q_1, \ldots, q_n$ and each a_i is of the form $t_i \bowtie_i d_i$ where t_i is an offset-free linear term.

Lemma A.2. *If there exists an $\alpha > 0$ such that for all $\beta \in (0, \alpha)$ the following holds $M_{a_\delta,\beta} \models_{\mathbb{R}} \bigwedge_i p_i$, then there exists a real value $\eta \geqslant 1$ such that for all $\zeta \geqslant \eta$ that $a_{\zeta\delta} \models_{\delta\mathbb{R}} \bigwedge_i q_i$.*

Proof. This follows directly from Lemma A.1. Suppose for some a_δ there exists an $\alpha > 0$ such that for all $\beta \in (0, \alpha)$, $M_{a_\delta,\beta} \models_{\mathbb{R}} \bigwedge p_i$. Then $M_{a_\delta,\beta} \models_{\mathbb{R}} p_i$. By the previous lemma, there exists a real value $\eta_i \geqslant 1$ such that for any $\zeta \geqslant \eta_i$ that $a_{\zeta\delta} \models_{\delta\mathbb{R}} q_i$. Let $\eta = \max \eta_i$ and so $\eta \geqslant 1$. Then for all $\zeta \geqslant \eta$, $a_{\zeta\delta} \models_{\delta\mathbb{R}} q_i$ holds. □

A.2.3 Translating $\mathbb{R}$ Assignments to $\delta\mathbb{R}$ Assignments

Revisiting Lemma (2.7). *Suppose $M \models_{\mathbb{R}} p$. Let q be a delta-encoding of the literal p and the assignment a_δ map all variables x_i in p to $\langle x_i^M, 0\rangle$. Then $a_\delta \models_{\delta\mathbb{R}} q$ where is the delta-encoding of a literal p.*

Proof. Suppose that $M \models_{\mathbb{R}} p$ and a_δ is an assignment described as above. The literal p has the form $t \bowtie d$. The evaluation of t under a_δ is $t^M + 0\delta$ where t^M is the evaluation of t under M. Then $t^M \bowtie d$ as M is satisfying.

- If $\bowtie$ is either $\geqslant$, $\leqslant$, $=$, or $\neq$, then q has the form $t \bowtie_\delta d + 0\delta$. Then as $t^M \bowtie d$ holds, $t^M + 0\delta \bowtie_\delta d + 0\delta$.
- If $\bowtie$ is $>$, then q has the form $t \bowtie_\delta d + e\delta$ where $e > 0$. Then as $t^M > d$ then $t^M + 0\delta >_\delta d + d\delta$.
- If $\bowtie$ is $<$, then q has the form $t \bowtie_\delta d + e\delta$ where $e < 0$. Then as $t^M < d$ then $t^M + 0\delta <_\delta d + e\delta$.

Thus in all cases $a_\delta \models_{\delta\mathbb{R}} q$ holds. □

A.2.4 Delta Entailment

Revisiting Lemma (Lemma 2.8 Revisited). *If* $\bigwedge q_i \models_{\delta\mathbb{R}} q$, *then* $\bigwedge p_i \models_{\mathbb{R}} p$ *where* q_i *is an admissible delta-encoding of the literal* p_i.

Proof. Assume that $\bigwedge q_i \models_{\delta\mathbb{R}} q$. This means that for all assignments a_δ, if $a_\delta \models_{\delta\mathbb{R}} \bigwedge q_i$ holds, then $a_\delta \models_{\delta\mathbb{R}} q$ holds. Let M be any interpretation satisfying $\bigwedge p_i$. Let a_δ be an assignment that maps x_i to $\langle x_i^M, 0\rangle$. Then $a_\delta \models_{\delta\mathbb{R}} q_i$ for each q_i. Thus $a_\delta \models_{\delta\mathbb{R}} q$. Structurally, $q \equiv t \bowtie_\delta d + e\delta$. The evaluation of t by M is t^M. Thus the evaluation of t under a_δ is $\langle t^M, 0\rangle$. For all cases, we know that $t^M + 0\delta \bowtie_\delta d + e\delta$ holds.

- Suppose that $e < 0$. Then $\bowtie_\delta$ must be $\leqslant_\delta$ and p has the form $t < d$. Because q is satisfied by a_δ, we have $t^M + 0\delta \leqslant_\delta d + e\delta$. To satisfy this $t^M \leqslant d$ and $t^M \neq d$ as $e < 0$. Thus $t^M < d$ and $M \models_{\mathbb{R}} p$.
- Suppose that $e = 0$. Then either $\bowtie_\delta$ is $=_\delta$, $\neq_\delta$, $\geqslant_\delta$, or $\leqslant_\delta$. Because q is satisfied by a_δ, we must have $t^M + 0\delta \bowtie_\delta d + 0\delta$. To satisfy this, $t^M \bowtie d$. Thus $M \models_{\mathbb{R}} p$.
- Suppose that $e > 0$. Then $\bowtie_\delta$ must be $\geqslant_\delta$ and p has the form $t > d$. Because q is satisfied by a_δ, we have $t^M + 0\delta \geqslant_\delta d + e\delta$. To satisfy this $t^M \geqslant d$ and $t^M \neq d$ as $e > 0$. Thus $t^M > d$ and $M \models_{\mathbb{R}} p$.

□

The reverse direction of the previous lemma.

Lemma A.3. *Let each* q_i *and* q *is an admissible delta-encoding of the literals* p_i *and* p *which have the form* $t \bowtie d$ *where* t_i *is an offset-free linear term. If* $\bigwedge p_i \models_{\mathbb{R}} p$, *then for every* $a_\delta \models_{\delta\mathbb{R}} \bigwedge q_i$ *there exists an* $\eta \geqslant 1$ *such that for all* $\zeta \geqslant \eta$ *that* $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$.

Proof. Suppose that $\bigwedge p_i \models_{\mathbb{R}} p$.

Let a_δ be an extended assignment that satisfies $\bigwedge q_i$. Then by Lemma 2.6 there exists an $\alpha > 0$ such that for all $\beta \in (0, \alpha)$ that $M_{a_\delta,\beta} \models_{\mathbb{R}} \bigwedge p_i$. Thus $M_{a_\delta,\beta} \models_{\mathbb{R}} p$ as $\bigwedge p_i$ entails p. Thus there exists an $\alpha > 0$ such that for all $\beta \in (0, \alpha)$ that $M_{a_\delta,\beta} \models_{\mathbb{R}} p$. Then by Lemma A.1 there exists a real value $\eta \geqslant 1$ such that for all $\zeta \geqslant \eta$ that $a_{\zeta\delta} \models_{\delta\mathbb{R}} q$. □

A.2.5 Externally Checkable Proof Witnesses

The vector y, the constraints that form A, and $\mathcal{L}$ and $\mathcal{U}$ can be converted to a more traditional witness of the entailment of the inequality in the form of Farkas' lemma. The main use of such proofs in SMT is the derivation of interpolants (which are outside of the scope of this thesis) [28]. Such witnesses may be sent to an external proof checking tools (such as LFSC [91]).

Let B be the $4n \times n$ matrix derived by converting the constraints $A\mathcal{X} = 0$ and $l \leqslant \mathcal{X} \leqslant u$ into only less than or equal to inequalities, and b be the corresponding right-hand-side.

$$B = \begin{bmatrix} -A \\ A \\ I \\ -I \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 0 \\ l \\ -u \end{bmatrix} \qquad B\mathcal{X} \geqslant b$$

We can then construct a $4n$-dimensional $\mathbb{R}$ row vector y' from y, z and $\mathcal{L}$ and $\mathcal{U}$, that witnesses the correctness of the entailment of $-\sum_{k \in \mathcal{F}} z_k x_k \geqslant \gamma$. For all i in the range $[1, n]$ we define the $kn + i$ component of y' as k ranges from 0 to 3:

$$y'_{0n+i} = \begin{cases} y_i & y_i > 0 \\ 0 & y_i \leqslant 0 \end{cases}$$

$$y'_{1n+i} = \begin{cases} -y_i & y_i < 0 \\ 0 & y_i \geqslant 0 \end{cases}$$

$$y'_{2n+i} = \begin{cases} z_i & i \in \mathcal{L} \\ 0 & i \notin \mathcal{L} \end{cases}$$

$$y'_{3n+i} = \begin{cases} -z_i & i \in \mathcal{U} \\ 0 & i \notin \mathcal{U} \end{cases}$$

We leave the correctness of this construction to the reader. These objects may be made more compact by removing the rows where y'_j is 0. See [28] for an extensive treatment of such proof witnesses for interpolants.

A.2.6 Tableau

The proof for Lemma 2.19 relies on a well known property of linear algebra. Let the $n \times n$ matrix M be a finite product of elementary operations $E_1, \dots, E_k$, let A be an $n \times n$ matrix, and X and B be n-dimensional column vectors.

Lemma A.4. *Then* $AX = B$ *iff* $MAX = MB$ *[4, Proposition 1.2.10].*

Let π be the permutation over $\mathcal{X}$ such that all basic variables are ordered before all non-basic variables. Let ρ be the permutation matrix for π. Let G be the result of applying Gaussian elimination to $\rho\, A\, \rho$. Let F be the composition of the elementary row operations performed by Gaussian elimination reducing $\rho\, A\, \rho$ to G.

Lemma A.5. $G = -\rho T \rho$.

Proof. Let $C = -\rho T \rho$ or the negative row and column permutation of T by π, $-T_{\pi(i),\pi(j)}$. Let d be the dimensions of A ($d = |\mathcal{B}|$). Let H be the composition of the elementary row operations performed by pivoting from A throughout execution to construct T.

$$C = -\rho H\, A\, \rho \qquad G = F \rho\, A\, \rho$$

By the tableau normal form for T and the row echelon form for Gaussian elimination, C and G can be decomposed into the matrices

$$C = \begin{bmatrix} I & X \\ 0 & 0 \end{bmatrix} \qquad G = \begin{bmatrix} I & Y \\ 0 & 0 \end{bmatrix}$$

where X and Y are $(n-d) \times (n-d)$ matrices. The matrices C and G are equal iff X and Y are equal. Suppose for contradiction that C and G are not equal. Then they must be not equal at some row i and column j such that i is in the range 1 to d, and j is in the range $d+1$ to n. Thus $\pi^{-1}(j) \in \mathcal{N}$ and $\pi^{-1}(i) \in \mathcal{B}$. Let α, β be an n-dimensional vectors such that

$$\alpha_\mu = \begin{cases} 1 & \mu = p^{-1}(j) \\ -C_{p(\mu),j} & \mu \in \mathcal{B} \\ 0 & \perp \end{cases} \qquad \beta_\nu = \begin{cases} 1 & \nu = j \\ -C_{\nu,j} & \nu \in [1,d] \\ 0 & \perp \end{cases}$$

The intuition behind the construction of α is that exactly 1 non-basic variable, $\pi^{-1}(j)$, has been set to an assignment of 1 with the rest of the non-basic variables set to 0. The basic variables are then computed from this. By construction, $\beta = \rho\alpha$. Thus $-\rho H\, A\, \beta = 0$ while $F\rho\, A\, \beta$ cannot be 0 on row i. By A.4, the solutions of $A\, X = 0$ must be preserved by right multiplication of elementary operations. Thus both $A\, \beta = 0$ and $A\, \beta \neq 0$. This is a contradiction and $C = G$. □

Theorem A.6. *The size of* T *and* a *is polynomial in the size of* $\Phi_{\mathcal{A}}$ *where* $\Phi_{\mathcal{A}}$ *is the set of all atoms in the input formula* ϕ *and lemmas sent to the SAT solver.*

Proof. The complexity of A is polynomial in the size of $\Phi_{\mathcal{A}}$ by construction. The complexity of T polynomial in the size of A by Lemma 2.19. The complexity of a_j for $j \in \mathcal{N}$ is determined by S as it is either equal to 0 or was set to some bound when it was set so it is equal to some delta rational for some encoding of a literal of an atom in $\Phi_{\mathcal{A}}$. (It is not necessarily equal to l or u as bounds may be popped. We are implicitly assuming here that the selected d values for strict inequalities for l and u have polynomial complexity.) By combining the complexity of a_j for $j \in \mathcal{N}$ and T, the assignment of basic variables must to be polynomial as well if the invariant (I3) holds. □

A.2.7 Direct Proof of Sum-of-Infeasibility Conflicts

For all $i \in \mathcal{B}$, let y_i be the row vector s.t. $T_i = y_i A$. Lemma 2.16 gives details on extracting y_i. Let z^S be the row vector $\sum_{i \in S} \lambda_i y_i A$ for a subset S of E. The k'th element of z^S is denoted z^S_k. Let $\mathcal{L}^S = \left\{k | z^S_k > 0\right\}$ and $\mathcal{U}^S = \left\{k | z^S_k < 0\right\}$. Let $\mathfrak{L}^S = \left\{x_k \geqslant_\delta l_k | k \in \mathcal{L}^S\right\}$, $\mathfrak{U}^S = \left\{x_k \leqslant_\delta u_k | k \in \mathcal{U}^S\right\}$, and $\mathfrak{R}^S = \left\{A_k \cdot \mathcal{X} =_\delta 0 | z^S_k \neq 0, A_k \neq 0\right\}$.

Lemma A.7. *If* $V_i > 0$ *for all* $i \in S$*, and* $\mathcal{F}_{act}(f^S) = \varnothing$*, then* $\mathfrak{L}^S \cup \mathfrak{U}^S \cup \mathfrak{R}^S \models_{\delta\mathbb{R}} 0 >_\delta \sum_{i \in S} V_i$.

Proof. The goal of the proof is to show that that Cor. 2.14 is applicable with $\gamma = \sum_{i \in S} V_i$. The row z^S is essentially the sum of infeasibilities row.

$$\begin{aligned} z^S &= \sum_{i \in S} \lambda_i y_i A = \sum_{i \in S} \lambda_i T_i \\ &= \sum_{i \in S} \lambda_i (\tau_i - e_i) = \sum_{i \in S} \lambda_i \tau_i + \sum_{i \in S} \lambda_i e_i \\ &= f^S - \sum_{i \in S} \lambda_i e_i \end{aligned}$$

The constraints in $\mathfrak{R}^S$ ensure that $z^S \mathcal{X} = 0$. By $\mathcal{F}_{act}(f^S) = \varnothing$, then

$$\mathcal{L}_{act}(f^S) \cup \mathcal{U}_{act}(f^S) = \left\{k | f^S_k \neq 0, k \in \mathcal{N}\right\}.$$

Thus for all $k \in \mathfrak{L}^S \cap \mathcal{N}$, then $a_k = l_k$ and for all $k \in \mathfrak{U}^S \cap \mathcal{N}$, then $a_k = u_k$. As each $i \in S \subseteq E$ is basic, it's coefficient (-1) cannot cancel with any other row and $z^S_i = -\lambda_i$. So for all $i \in \mathfrak{L}^S \cap \mathcal{B}$, then $\lambda_i = -1$, $z^S_i > 0$, $a_i < l_i$ and $V_i = l_i - a_i$. For all $i \in \mathfrak{U}^S \cap \mathcal{B}$, then $\lambda_i = +1$, $a_k > u_k$ and $V_i = a_i - u_i$.

By definition the value of γ for $\mathfrak{L}^S$ and $\mathfrak{U}^S$ is

$$\gamma = \sum_{j \in \mathfrak{L}^S \cap \mathcal{N}} f^S_j l_j + \sum_{k \in \mathfrak{U}^S \cap \mathcal{N}} f^S_k u_k + \sum_{i \in \mathfrak{L}^s \cap \mathcal{B}} l_i + \sum_{i \in \mathfrak{U}^s \cap \mathcal{B}} -u_i.$$

The value of $z^E \cdot a = 0$ by the invariants on the tableau. Note that $-\lambda_i a_i = a_i = l_i - V_i$ and $-\lambda_i a_i = -a_i = -u_i - V_i$.

$$\begin{aligned} 0 &= f^S \cdot a - \sum_{i \in S} \lambda_i a_i \\ &= \sum_{j \in \mathfrak{L}^S \cap \mathcal{N}} f^S_j l_j + \sum_{k \in \mathfrak{U}^S \cap \mathcal{N}} f^S_k u_k + \sum_{i \in \mathfrak{L}^s \cap \mathcal{B}} l_i - V_i + \sum_{i \in \mathfrak{U}^s \cap \mathcal{B}} -u_i - V_i \\ \sum_{i \in E} V_i &= \gamma \end{aligned}$$

As $V(\mathcal{X}) = \sum_{i \in E} V_i >_\delta 0$, Cor. 2.14 is directly applicable. $\square$

Bibliography

[1] Achterberg, T. SCIP: solving constraint integer programs. *Mathematical Programming Computation 1*, 1 (2009), 1–41. 134

[2] Achterberg, T., Koch, T., and Martin, A. MIPLIB 2003. *Operations Research Letters 34*, 4 (2006), 361–372. 133

[3] Applegate, D. L., Cook, W., Dash, S., and Espinoza, D. G. Exact solutions to linear programming problems. *Operations Research Letters 35*, 6 (2007), 693–699. 124, 137

[4] Artin, M. *Algebra*. Pearson Education, 2011. 153

[5] Badros, G., Borning, A., and Stuckey, P. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction 8*, 4 (2001), 267–306. 85

[6] Barrett, C., and Berezin, S. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification* (2004), vol. 3114 of *LNCS*, pp. 515–518. 19

[7] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *Computer Aided Verification* (2011), vol. 6806 of *LNCS*, pp. 171–177. 19, 35, 133

[8] Barrett, C., de Moura, L., Ranise, S., Stump, A., and Tinelli, C. The SMT-LIB Initiative and the Rise of SMT. In *Hardware and Software: Verification and Testing* (2011), vol. 6504 of *LNCS*, pp. 3–3. 2

[9] Barrett, C., Dill, D. L., and Levitt, J. R. Validity Checking for Combinations of Theories with Equality. In *Formal Methods in Computer-Aided Design* (1996), vol. 1166 of *LNCS*, pp. 187–201. 19

[10] Barrett, C., Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Splitting on Demand in SAT Modulo Theories. In *Logic for Programming, Artificial Intelligence, and Reasoning* (2006), vol. 4246 of *LNCS*, pp. 512–526. 23

[11] BARRETT, C., SEBASTIANI, R., SESHIA, S., AND TINELLI, C. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, ch. 26, pp. 825–885. 18, 19

[12] BARRETT, C., STUMP, A., AND TINELLI, C. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2010. 19, 101

[13] BARRETT, C., AND TINELLI, C. CVC3. In *Computer Aided Verification* (2007), vol. 4590 of *LNCS*, pp. 298–302. 19

[14] BIERE, A. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation 4*, 2-4 (2008), 75–97. 16

[15] BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems* (1999), vol. 1579 of *LNCS*, pp. 193–207. 2

[16] BOBOT, F., CONCHON, S., CONTEJEAN, E., IGUERNELALA, M., LESCUYER, S., AND MEBSOUT, A. The Alt-Ergo Automated Theorem Prover. `http://alt-ergo.lri.fr`. 134

[17] BOBOT, F., CONCHON, S., CONTEJEAN, E., IGUERNELALA, M., MAHBOUBI, A., MEBSOUT, A., AND MELQUIOND, G. A Simplex-Based Extension of Fourier-Motzkin for Solving Linear Integer Arithmetic. In *International Joint Conference on Automated Reasoning* (2012), vol. 7364 of *LNCS*, pp. 67–81. 134

[18] BOFILL, M., NIEUWENHUIS, R., OLIVERAS, A., RODRÍGUEZ-CARBONELL, E., AND RUBIO, A. The Barcelogic SMT Solver. In *Computer Aided Verification* (2008), vol. 5123 of *LNCS*, pp. 294–298. 35, 120

[19] BOYD, S., AND VANDENBERGHE, L. *Convex Optimization*. Cambridge University Press, 2004. 47, 85, 89, 91

[20] BOZZANO, M., BRUTTOMESSO, R., CIMATTI, A., JUNTTILA, T., RANISE, S., ROSSUM, P. V., AND SEBASTIANI, R. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *Computer Aided Verification* (2005), vol. 3576 of *LNCS*, pp. 335–349. 26

[21] BRADLEY, A. R., AND MANNA, Z. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, 2007. 89

[22] BRUTTOMESSO, R., FULVIO ROLLINI, S., SHARYGINA, N., AND TSITOVICH, A. OpenSMT Source Code r64. `http://opensmt.googlecode.com/svn/trunk/src/tsolvers/lrasolver/LRASolver.C`. 79

[23] Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. The OpenSMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (2011), vol. 6605 of *LNCS*, pp. 150–153. 35, 120

[24] Bruttomesso, R. B., Cok, D., and Griggio, A. SMT-COMP 2012. `http://smtcomp.sourceforge.net/2012/`, 2012. 101

[25] Bryant, R. E., Grumberg, O., Sifakis, J., and Vardi, M. Y. 2009 CAV award announcement. *Formal Methods in System Design 36*, 3 (2010), 195–197. 2

[26] Christ, J., Hoenicke, J., and Nutz, A. SMTInterpol: an interpolating SMT solver. In *International SPIN Workshop on Model Checking of Software* (2012), vol. 7385 of *LNCS*, pp. 248–254. 35

[27] Cimatti, A., Griggio, A., Schaafsma, B., and Sebastiani, R. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (2013), vol. 7795 of *LNCS*, pp. 93–107. 35, 134

[28] Cimatti, A., Griggio, A., and Sebastiani, R. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM Transactions on Computational Logic 12*, 1 (2010), 7:1–7:54. 59, 152

[29] Cimatti, A., Griggio, A., and Sebastiani, R. Computing Small Unsatisfiable Cores in Satisfiability Modulo Theories. *Journal of Artificial Intelligence Research 40*, 1 (2011), 701–728. 21

[30] Clarke, E. M. The Birth of Model Checking. In *25 Years of Model Checking*. Springer-Verlag, 2008, pp. 1–26. 2

[31] Collins, G. E. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages* (1975), vol. 33 of *LNCS*, pp. 134–183. 36

[32] Cook, W., Dash, S., Fukasawa, R., and Goycoolea, M. Numerically Safe Gomory Mixed-Integer Cuts. *INFORMS Journal on Computing 21*, 4 (2009), 641–649. 136

[33] Cook, W., Koch, T., Steffy, D. E., and Wolter, K. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation 5*, 3 (2013), 305–344. 124, 134, 137

[34] Cooper, D. C. Theorem proving in arithmetic without multiplication. *Machine Intelligence 7* (1972), 91–99. 105

[35] Darwiche, A., and Pipatsrisawat, K. Complete Algorithms. In *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, ch. 3, pp. 99–130. 15

[36] Davis, M., Logemann, G., and Loveland, D. A Machine Program for Theorem-proving. *Communications of the ACM 5*, 7 (1962), 394–397. 15

[37] Davis, M., and Putnam, H. A Computing Procedure for Quantification Theory. *Journal of the ACM 7*, 3 (1960), 201–215. 15

[38] de Moura, L., and Bjørner, N. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (2008), vol. 4963 of *LNCS*, pp. 337–340. 35, 134

[39] de Moura, L., Bjørner, N., and Wintersteiger, C. Z3 Source Code v4.3.1. 79

[40] de Moura, L., and Jovanović, D. A Model-Constructing Satisfiability Calculus. In *Verification, Model Checking, and Abstract Interpretation* (2013), vol. 7737 of *LNCS*, pp. 1–12. 36, 138

[41] de Moura, L., and Passmore, G. O. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics*. Springer, 2013, pp. 15–44. 29

[42] de Oliveira, D. C. B., and Monniaux, D. Experiments on the feasibility of using a floating-point simplex in an SMT solver. In *Workshop on Practical Aspects of Automated Reasoning* (2012), CEUR Workshop Proceedings, pp. 19–28. 120, 121, 124

[43] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM 52*, 3 (2005), 365–473. 13, 85

[44] Dillig, I., Dillig, T., and Aiken, A. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In *Computer Aided Verification* (2009), vol. 5643 of *LNCS*, pp. 233–247. 110, 135

[45] Dutertre, B. Yices2.2. In *Computer Aided Verification* (2014), vol. 8559 of *LNCS*, pp. 737–744. 35, 134

[46] Dutertre, B., and de Moura, L. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification* (2006), vol. 4144 of *LNCS*, pp. 81–94. 34, 124, 133

[47] DUTERTRE, B., AND DE MOURA, L. Integrating Simplex with DPLL(T). Tech. Rep. SRI-CSL-06-01, Computer Science Laboratory, SRI International, 2006. 38, 47, 77, 78, 84, 112, 115, 131, 133

[48] DUTERTRE, B., AND DE MOURA, L. The YICES SMT Solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, 2006. 35

[49] ENDERTON, H. *A Mathematical Introduction to Logic*, second ed. Academic Press, 2001. 8, 104, 105

[50] FARKAS, G. A Fourier-féle mechanikai elv alkamazásai (Hungarian). *Mathematikai és Természettudományi Értesítő 12* (1894), 457–472. reference from Schrijver's Combinatorial Optimization textbook. 55

[51] FAURE, G., NIEUWENHUIS, R., OLIVERAS, A., AND RODRÍGUEZ-CARBONELL, E. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In *Theory and Applications of Satisfiability Testing* (2008), vol. 4996 of *LNCS*, pp. 77–90. 120, 121

[52] FISCHER, M. J., AND RABIN, M. O. Super-exponential complexity of presburger arithmetic. Tech. rep., Massachusetts Institute of Technology, 1974. 105

[53] FORREST, J. J., AND GOLDFARB, D. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming 57*, 1-3 (1992), 341–374. 89

[54] GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification* (2004), vol. 3114 of *LNCS*, pp. 175–188. 18

[55] GASS, S. I. George B. Dantzig. In *Profiles in Operations Research*, vol. 147 of *International Series in Operations Research & Management Science*. Springer, 2011, pp. 217–240. 84

[56] GILL, P. E., MURRAY, W., AND WRIGHT, M. H. *Numerical linear algebra and optimization*, vol. 1. Addison-Wesley Publishing Company, 1991. 36, 47, 84, 85, 89, 91, 95

[57] GÖDEL, K. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, reprint, the original paper publised in 1931 ed. Dover Publications, 1992. 104

[58] GOMORY, R. E. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society 64*, 5 (1958), 275–278. 112

[59] GOMORY, R. E. Solving Linear Programming Problems in Integers. *Proceedings of Symposia in Applied Mathematics 10* (1960), 211–215. 112

[60] GRIGGIO, A. *An Effective SMT Engine for Formal Verification*. PhD thesis, DISI - University of Trento, December 2009. 79

[61] GRIGGIO, A. A Practical Approach to Satisability Modulo Linear Integer Arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation 8*, 1/2 (2012), 1–27. 109, 110

[62] GRUMBERG, O., VARDI, M., SIFAKIS, J., AND ALUR, R. 2010 cav award announcement. *Formal Methods in System Design 40*, 2 (2012), 117–120. 2

[63] HARRIS, P. M. J. Pivot selection methods of the Devex LP code. *Mathematical Programming 5*, 1 (1973), 1–28. 89

[64] HARRISON, J. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. 13, 105

[65] HODGES, W. *A Shorter Model Theory*. Cambridge University Press, New York, NY, USA, 1997. 12, 34, 36

[66] HUANG, J. The Effect of Restarts on the Efficiency of Clause Learning. In *International Joint Conference on Artifical Intelligence* (2007), pp. 2318–2323. 16

[67] JOVANOVIĆ, D. personal communication, 2013. 110

[68] JOVANOVIĆ, D., AND BARRETT, C. Being careful about theory combination. *Formal Methods in System Design 42*, 1 (2013), 67–90. 26

[69] JOVANOVIĆ, D., AND DE MOURA, L. Cutting to the Chase Solving Linear Integer Arithmetic. In *Conference on Automated Deduction* (2011), pp. 338–353. 134, 135

[70] JOVANOVIĆ, D., AND DE MOURA, L. Solving Non-linear Arithmetic. In *International Joint Conference on Automated Reasoning* (2012), vol. 7364 of *LNCS*, pp. 339–354. 36

[71] JUNKER, U. QuickXplain: Conflict Detection for Arbitrary Constraint Propagation Algorithms. In *IJCAI-01 Workshop on Modelling and Solving Problems with Constraints* (2001). 97

[72] JUNKER, U. QuickXplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artifical Intelligence (AAAI)* (2004), pp. 167–172. 97

[73] KING, T., BARRETT, C., AND DUTERTRE, B. Simplex with sum of infeasibilities for SMT. In *Formal Methods in Computer-Aided Design* (2013), pp. 189–196. 4, 85, 97, 133, 134

[74] KING, T., BARRETT, C., AND TINELLI, C. Leveraging linear and mixed integer programming for SMT. In *Formal Methods in Computer-Aided Design* (2014), p. to appear. 5, 120

[75] KONEV, B., AND LISITSA, A. A SAT Attack on the Erds Discrepancy Conjecture. In *Theory and Applications of Satisfiability Testing* (2014), vol. 8561, pp. 219–226. 2

[76] LATENDRESSE, M., KRUMMENACKER, M., TRUPP, M., AND KARP, P. D. Construction and completion of flux balance models from pathway databases. *Bioinformatics 28* (2012), 388–96. 101

[77] LI, Y., ALBARGHOUTHI, A., KINCAID, Z., GURFINKEL, A., AND CHECHIK, M. Symbolic Optimization with SMT Solvers. *ACM SIGPLAN Notices 49*, 1 (2014), 607–618. 136

[78] MAKHORIN, A. GNU Linear Programming Kit, Version 4.52. `http://www.gnu.org/software/glpk/glpk.html`, June 2012. 133, 134

[79] MARCHAND, H., AND WOLSEY, L. A. Aggregation and mixed integer rounding to solve mips, 1998. 112, 131

[80] MARCHAND, H., AND WOLSEY, L. A. Aggregation and Mixed Integer Rounding to Solve MIPS. *Operations Research 49*, 3 (2001), 363–371. 131

[81] MARQUES-SILVA, J., AND SAKALLAH, K. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48*, 5 (May 1999), 506–521. 17, 18

[82] MATIYASEVICH, Y. V. *Hilbert's Tenth Problem*. MIT Press, 1993. 105

[83] MCMILLAN, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 2

[84] MEYER, R. On the existence of optimal solutions to integer and mixed-integer programming problems. *Mathematical Programming 7*, 1 (1974), 223–235. 118

[85] MONNIAUX, D. On using floating-point computations to help an exact linear arithmetic decision procedure. In *Computer Aided Verification* (2009), no. 5643 in LNCS, pp. 570–583. 124

[86] MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., ZHANG, L., AND MALIK, S. Chaff: engineering an efficient sat solver. In *Design Automation Conference* (2001), pp. 530–535. 16

[87] NELSON, G., AND OPPEN, D. C. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems 1*, 2 (1979), 245–257. 26

[88] NEMHAUSER, G. L., AND WOLSEY, L. A. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1988. 112, 123, 131

[89] NEUMAIER, A., AND SHCHERBINA, O. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming 99*, 2 (2004), 283–296. 124, 136

[90] NIEWENHUIS, R., OLIVERAS, A., AND TINELLI, C. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM 53*, 6 (2006), 937–977. 18, 19, 138

[91] OE, D., REYNOLDS, A., AND STUMP, A. Fast and Flexible Proof Checking for SMT. In *International Workshop on Satisfiability Modulo Theories* (2009), pp. 6–13. 58, 152

[92] OPPEN, D. C. A 222pn upper bound on the complexity of Presburger Arithmetic. *Journal of Computer and System Sciences 16*, 3 (1978), 323–332. 105

[93] PAPADIMITRIOU, C. H. On the complexity of integer programming. *Journal of the ACM 28*, 4 (1981), 765–768. 105

[94] PIPATSRISAWAT, K., AND DARWICHE, A. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Theory and Applications of Satisfiability Testing* (2007), vol. 4501 of *LNCS*, pp. 294–299. 16

[95] RANISE, S., AND TINELLI, C. The SMT-LIB standard: Version 1.2. Tech. rep., The University of Iowa, 2006. 19

[96] REYNOLDS, A. J. *Finite Model Finding in Satisfiability Modulo Theories*. PhD thesis, The University of Iowa, 2013. 13, 23

[97] RUESS, H., AND SHANKAR, N. Solving Linear Arithmetic Constraints. Tech. Rep. SRI-CSL-04-01, SRI International, 2004. 76, 85

[98] SCHRIJVER, A. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1989. 47, 55, 69, 84, 89, 95, 112

[99] SEBASTIANI, R. Lazy Satisability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation 3*, 3-4 (2007), 141–224. 18, 19

[100] SEBASTIANI, R., AND TOMASI, S. Optimization in SMT with LA(Q) Cost Functions. In *International Joint Conference on Automated Reasoning* (2012), vol. 7364 of *LNCS*, pp. 484–498. 136

[101] SILVA, J. P. M., LYNCE, I., AND MALIK, S. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, ch. 4, pp. 131–153. 17, 18, 130

[102] SOFRONIE-STOKKERMANS, V. Hierarchic Reasoning in Local Theory Extensions. In *Conference on Automated Deduction* (2005), vol. 3632 of *LNCS*, pp. 219–234. 13

[103] STUMP, A., BARRETT, C., AND DILL, D. L. CVC: A Cooperating Validity Checker. In *Computer Aided Verification* (2002), vol. 2404 of *LNCS*, pp. 500–504. 19

[104] STUMP, A., SUTCLIFFE, G., AND TINELLI, C. StarExec: a Cross-Community Infrastructure for Logic Solving. In *International Joint Conference on Automated Reasoning* (2014), vol. 8562 of *LNCS*, pp. 367–373. 133

[105] SUTCLIFFE, G., AND SUTTNER, C. The TPTP Problem Library. *Journal of Automated Reasoning 21*, 2 (1998), 177–203. 19

[106] TARSKI, A. A decision method for elementary algebra and geometry. Tech. rep., Rand Corporation, 1951. 36

[107] TINELLI, C. SMT-LIB 2 Theory Ints. `http://smtlib.cs.uiowa.edu/theories/Ints.smt2`, 2010. Accessed: 2014-07-29. 105

[108] TINELLI, C., AND HARANDI, M. T. A new correctness proof of the nelson-oppen combination procedure. In *Frontiers of Combining Systems* (1996), vol. 3 of *Applied Logic Series*, pp. 103–119. 26

[109] TSEITIN, G. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning*, Symbolic Computation. Springer, 1983, pp. 466–483. 14

[110] WEISPFENNING, V. Mixed Real-integer Linear Quantifier Elimination. In *International Symposium on Symbolic and Algebraic Computation* (1999), pp. 129–136. 106

[111] YU, Y., AND MALIK, S. Lemma Learning in SMT on Linear Constraints. In *Theory and Applications of Satisfiability Testing* (2006), vol. 4121 of *LNCS*, pp. 142–155. 120